Gtk4 tutorial for beginners

Toshio Sekiya

abstract

Contents of this Repository This tutorial illustrates how to write C programs with the GTK 4 library. It focuses on beginners so the contents are limited to the basics. The table of contents is at the end of this abstract.

- Section 3 to 23 describes the basics, with the example of a simple editor `tfe` (Text File Editor).
- Section 24 to 27 describes GtkDrawingArea.
- Section 28 describes Drag and Drop.
- Section 29 to 33 describes the list model and the list view (GtkListView, GtkGridView and GtkColumnView). It also describes GtkExpression.

The latest version of the tutorial is located at GTK4-tutorial GitHub repository. You can read it directly without download.

There's a GitHub Page which is the HTML version of the tutorial.

GTK 4 Documentation Please refer to GTK 4 API Reference and GNOME Developer Documentation Website for further information.

These websites were opened in August of 2021. The old documents are located at GTK Reference Manual and GNOME Developer Center.

If you want to know about GObject and the type system, please refer to GObject tutorial. GObject is the base of GTK 4, so it is important for developers to understand it as well as GTK 4.

Contribution This tutorial is still under development and unstable. Even though the codes of the examples have been tested on GTK 4 (version 4.10.1), bugs may still exist. If you find any bugs, errors or mistakes in the tutorial and C examples, please let me know. You can post it to GitHub issues. You can also post updated files to pull request. One thing you need to be careful is to correct the source files, which are under the 'src' directory. Don't modify the files under `gfm` or `html` directories. After modifying some source files , run `rake` to create GFM (GitHub Flavoured Markdown) files and run `rake html` to create HTML files.

If you have a question, feel free to post it to `issue`. All questions are helpful and will make this tutorial get better.

How to get Gtk 4 tutorial with HTML or PDF format If you want to get HTML or PDF format tutorial, make them with `rake` command, which is a ruby version of make. Type `rake html` for HTML. Type `rake pdf` for PDF. An appendix "How to build GTK 4 Tutorial" describes how to make them.

License The license of this repository is written in Section1. In short,

- GFDL1.3 for documents
- GPL3 for programs

Contents

1 Prerequisite and License

1.1 Prerequisite

1.1.1 GTK 4 on a Linux OS

This tutorial describes GTK 4 libraries. It is originally used on Linux with C compiler, but now it is used more widely, on Windows and MacOS, with Vala, Python, Ruby and so on. However, this tutorial describes only *C programs on Linux*.

If you want to try the examples in the tutorial, you need:

- PC with Linux distribution like Ubuntu or Debian.
- GCC.
- GTK 4 (version 4.10.1 or later).

The stable version of GTK is 4.10.1 at present (24/April/2023). The version 4.10 adds some new classes and functions and makes some classes and functions deprecated. Some example programs in this tutorial don't work on the older version.

1.1.2 Ruby and rake for making the document

This repository includes Ruby programs. They are used to make GFM (GitHub Flavoured Markdown) files, HTML files, Latex files and a PDF file.

You need:

- Linux.
- Ruby programming language. There are two ways to install. One is installing the distribution's package. The other is using rbenv and ruby-build. If you want to use the latest version of ruby, use rbenv.
- Rake.

1.2 License

GTK4-tutorial repository contains tutorial documents and programs such as converters, generators and controllers. All of them make up the 'GTK4-tutorial' package. This package is simply called 'GTK4-tutorial' in the following description.

GTK4-tutorial is free; you can redistribute it and/or modify it under the terms of the following licenses.

- The license of documents in GTK4-tutorial is the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License or, at your opinion, any later version. The documents are Markdown, HTML and image files. If you generate a PDF file by running `rake pdf`, it is also included by the documents.
- The license of programs in GTK4-tutorial is the GNU General Public License as published by the Free Software Foundation; either version 3 of the License or, at your option, any later version. The programs are written in C, Ruby and other languages.

GTK4-tutorial is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU License web pages for more details.

- GNU Free Documentation License
- GNU General Public License

The licenses above is effective since 15/April/2023. Before that, GPL covered all the contents of the GTK4-tutorial. But GFDL1.3 is more appropriate for documents so the license was changed. The license above is the only effective license since 15/April/2023.

2 Preparation

2.1 Installing GTK 4 into Linux distributions

This section describes how to install GTK 4 into Linux distributions.

There are two ways to install GTK 4.

- Install it from the distribution packages.
- Build it from the source files.

2.1.1 Installation from the distribution packages

This is the easiest way to install. I've installed GTK 4 packages (version 4.14.2) on Ubuntu 24.04 LTS.

```
$ sudo apt install libgtk-4-dev
```

It is important to install the development files package (libgtk-4-dev). Otherwise, you can't compile any GTK 4 based programs.

Fedora, Debian, Arch, Gentoo and OpenSUSE also have GTK 4 packages. See the website of your distribution for further information.

Package information for Arch, Debian/Ubuntu and Fedora is described in GTK website, Installing GTK from packages.

2.1.2 Installation from the source file

If you want to install a developing version of GTK 4, you need to build it from the source. See Compiling the GTK Libraries section in the GTK 4 API reference.

2.2 How to download this repository

There are two ways: zip and git. Downloading a zip file is the easiest way. However, if you use git and clone this repository, you can easily update your local repository by `git pull` command.

2.2.1 Download a zip file

- Run your browser and open this repository.
- Click on the green button with `<> Code`. Then a popup menu appears. Click on `Download ZIP` menu.
- Then the repository data is downloaded as a zip file into your download folder.
- Unzip the file.

2.2.2 Clone the repository

- Click on the green button with the label `<> Code`. Then a popup menu appears. The first section is `Clone` with three tabs. Click `HTTPS` tab and click on the copy icon, which is on the right of `https://github.com/ToshioCP/Gtk4-tutorial.git`.
- Run your terminal and type `git clone`, Ctrl+Shift+V. Then the line will be `git clone https://github.com/ToshioCP/Gtk4-tutorial.git`. Press the enter key.
- A directory `Gtk4-tutorial` is created. It is the copy of this repository.

2.3 Examples in the tutorial

Examples are under the `src` directory. For example, the first example of the tutorial is `pr1.c` and its pathname is `src/misc/pr1.c`. So you don't need to type the example codes by yourself.

3 GtkApplication and GtkApplicationWindow

3.1 GtkApplication

3.1.1 GtkApplication and g_application_run

People write programming codes to make an application. What are applications? Applications are software that runs using libraries, which are the OS, frameworks and so on. In GTK 4 programming, the GtkApplication is a program (or executable) that runs using Gtk libraries.

The basic way to write a GtkApplication is as follows.

- Create a GtkApplication instance.
- Run the application.

That's all. Very simple. The following is the C code representing the scenario above.

```
1  #include <gtk/gtk.h>
2
3  int
4  main (int argc, char **argv) {
5    GtkApplication *app;
6    int stat;
7
8    app = gtk_application_new ("com.github.ToshioCP.pr1", G_APPLICATION_DEFAULT_FLAGS);
9    stat =g_application_run (G_APPLICATION (app), argc, argv);
10   g_object_unref (app);
11   return stat;
12 }
```

The first line says that this program includes the header files of the Gtk libraries. The function `main` is a startup function in C language. The variable `app` is defined as a pointer to a GtkApplication instance. The function `gtk_application_new` creates a GtkApplication instance and returns a pointer to the instance. The GtkApplication instance is a C structure data in which the information of the application is stored. The arguments will be explained later. The function `g_application_run` runs an application that the instance defined. (We often say that the function runs `app`. Actually, `app` is not an application but a pointer to the instance of the application. However, it is simple and short, and probably no confusion occurs.)

Here I used the word `instance`. Instance, class and object are terminologies in Object Oriented Programming. I use these words in the same way. But, I will often use "object" instead of "instance" in this tutorial. That means "object" and "instance" is the same. Object is a bit ambiguous word. In a broad sense, object has wider meaning than instance. So, readers should be careful of the contexts to find the meaning of "object". In many cases, object and instance are interchangeable.

The function `gtk_application_new` has two parameters.

- Application ID (com.github.ToshioCP.pr1). It is used to distinguish applications by the system. The format is reverse-DNS. See GNOME Developer Documentation – Application ID for further information.
- Application flag (G_APPLICATION_DEFAULT_FLAGS). If the application runs without any arguments, the flag is G_APPLICATION_DEFAULT_FLAGS. Otherwise, you need other flags. See GIO API reference for further information.

Notice: If your GLib-2.0 version is older than 2.74, use `G_APPLICATION_FLAGS_NONE` instead of `G_APPLICATION_DEFAULT_FLAGS`. It is an old flag replaced by `G_APPLICATION_DEFAULT_FLAGS` and deprecated since version 2.74.

To compile this, run the following command. The string `pr1.c` is the filename of the C source code above.

If you've downloaded this repository, you don't need to create the file. There's the same file at `src/misc/pr1.c` in your local repository. All the example codes are under the `src` directory as well.

```
$ gcc `pkg-config --cflags gtk4` pr1.c `pkg-config --libs gtk4`
```

The C compiler gcc generates an executable file, `a.out`. Let's run it.

```
$ ./a.out

(a.out:5084): GLib-GIO-WARNING **: 09:52:04.236: Your application does not implement
g_application_activate() and has no handlers connected to the 'activate' signal.
It should do one of these.
$
```

Oh, it just produces an error message. This error message shows that the GtkApplication object ran, without a doubt. Now, let's think about what this message means.

3.1.2 Signals

The message tells us that:

1. The application doesn't implement `g_application_activate()`,
2. It has no handlers connected to the "activate" signal, and
3. You will need to solve at least one of these.

These two problems are related to signals. So, I will explain that first.

A signal is emitted when something happens. For example, a window is created, a window is destroyed and so on. The signal "activate" is emitted when the application is activated. (Activated is a bit different from started, but you can think of them both as almost the same so far.) If the signal is connected to a function, which is called a signal handler or simply handler, then the function is invoked when the signal is emitted.

The flow is like this:

1. Something happens.
2. If it's related to a certain signal, then the signal is emitted.
3. If the signal has been connected to a handler in advance, then the handler is invoked.

Signals are defined in objects. For example, the "activate" signal belongs to the GApplication object, which is a parent object of GtkApplication object.

The GApplication object is a child object of the GObject object. GObject is the top object in the hierarchy of all the objects.

```
GObject -- GApplication -- GtkApplication
<---parent                    --->child
```

A child object inherits signals, functions, properties and so on from its parent object. So, GtkApplication also has the "activate" signal.

Now we can solve the problem in `pr1.c`. We need to connect the "activate" signal to a handler. We use a function `g_signal_connect` which connects a signal to a handler.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app, gpointer *user_data) {
5    g_print ("GtkApplication␣is␣activated.\n");
6  }
7
8  int
9  main (int argc, char **argv) {
10   GtkApplication *app;
11   int stat;
12
13   app = gtk_application_new ("com.github.ToshioCP.pr2", G_APPLICATION_DEFAULT_FLAGS);
14   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
15   stat =g_application_run (G_APPLICATION (app), argc, argv);
16   g_object_unref (app);
17   return stat;
18 }
```

First, we define the handler `app_activate` which simply displays a message. The function `g_print` is defined in GLib and it's like a printf in the C standard library. In the function `main`, we add `g_signal_connect` before `g_application_run`. The function `g_signal_connect` has four arguments.

1. An instance to which the signal belongs.
2. The name of the signal.
3. A handler function (also called callback), which needs to be casted by `G_CALLBACK`.
4. Data to pass to the handler. If no data is necessary, NULL is given.

It is described in the GObject API Reference. Correctly, `g_signal_connect` is a macro (not a C function).

```
#define g_signal_connect (
  instance,
  detailed_signal,
  c_handler,
  data
)
```

You can find the description of each signal in the API reference manual. For example, "activate" signal is in the GApplication section in the GIO API Reference.

```
void
activate (
  GApplication* self,
  gpointer user_data
)
```

This is a declaration of the "activate" signal handler. You can use any name instead of "activate" in the declaration above. The parameters are:

- self is an instance to which the signal belongs.
- user_data is a data defined in the fourth argument of the `g_signal_connect` function. If it is NULL, then you can ignore and leave out the second parameter.

API reference manual is very important. You should see and understand it.

Let's compile the source file above (`pr2.c`) and run it.

```
$ gcc `pkg-config --cflags gtk4` pr2.c `pkg-config --libs gtk4`
$ ./a.out
GtkApplication is activated.
$
```

OK, well done. However, you may have noticed that it's painful to type such a long line to compile. It is a good idea to use shell script to solve this problem. Make a text file which contains the following line.

```
gcc `pkg-config --cflags gtk4` $1.c `pkg-config --libs gtk4`
```

Then, save it under the directory $HOME/bin, which is usually /home/(username)/bin. (If your user name is James, then the directory is /home/james/bin). And turn on the execute bit of the file. If the filename is `comp`, do like this:

```
$ chmod 755 $HOME/bin/comp
$ ls -log $HOME/bin
    ...  ...  ...
-rwxr-xr-x 1   62 May 23 08:21 comp
    ...  ...  ...
```

If this is the first time that you make a $HOME/bin directory and save a file in it, then you need to logout and login again.

```
$ comp pr2
$ ./a.out
GtkApplication is activated.
$
```

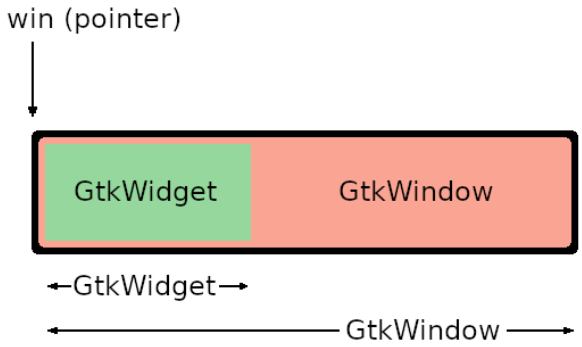


Figure 1: GtkWindow and GtkWidget

3.2 GtkWindow and GtkApplicationWindow

3.2.1 GtkWindow

A message "GtkApplication is activated." was printed out in the previous subsection. It was good in terms of a test of GtkApplication. However, it is insufficient because Gtk is a framework for graphical user interface (GUI). Now we go ahead with adding a window into this program. What we need to do is:

1. Create a GtkWindow.
2. Connect it to the GtkApplication.
3. Show the window.

Now rewrite the function `app_activate`.

```
1  static void
2  app_activate (GApplication *app, gpointer user_data) {
3    GtkWidget *win;
4
5    win = gtk_window_new ();
6    gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
7    gtk_window_present (GTK_WINDOW (win));
8  }
```

Widget is an abstract concept that includes all the GUI interfaces such as windows, labels, buttons, multi-line text, boxes and so on. And GtkWidget is a base object from which all the GUI objects derive.

```
parent <-----> child
GtkWidget -- GtkWindow
```

GtkWindow includes GtkWidget at the top of its object.

The function `gtk_window_new` is defined as follows.

```
GtkWidget *
gtk_window_new (void);
```

By this definition, it returns a pointer to GtkWidget, not GtkWindow. It actually creates a new GtkWindow instance (not GtkWidget) but returns a pointer to GtkWidget. However,the pointer points the GtkWidget and at the same time it also points GtkWindow that contains GtkWidget in it.

If you want to use `win` as a pointer to a GtkWindow type instance, you need to cast it.

```
(GtkWindow *) win
```

It works, but isn't usually used. Instead, `GTK_WINDOW` macro is used.

```
GTK_WINDOW (win)
```

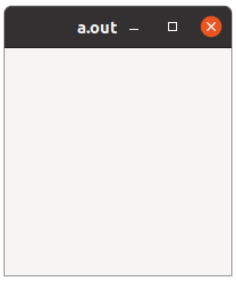


Figure 2: Screenshot of the window

The macro is recommended because it does not only cast the pointer but it also checks the type.

Connect it to the GtkApplication. The function `gtk_window_set_application` is used to connect GtkWindow to GtkApplication.

```
gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
```

You need to cast `win` to GtkWindow and `app` to GtkApplication with `GTK_WINDOW` and `GTK_APPLICATION` macro.

GtkApplication continues to run until the related window is destroyed. If you didn't connect GtkWindow and GtkApplication, GtkApplication destroys itself immediately. Because no window is connected to GtkApplication, GtkApplication doesn't need to wait anything. As it destroys itself, the GtkWindow is also destroyed.

Show the window. The function `gtk_window_present` presents the window to a user (shows it to the user).

GTK 4 changes the default widget visibility to on, so every widget doesn't need to change it to on. But, there's an exception. Top level window (this term will be explained later) isn't visible when it is created. So you need to use the function above to show the window.

You can use `gtk_widget_set_visible (win, true)` instead of `gtk_window_present`. But the behavior of these two is different. Suppose there are two windows win1 and win2 on the screen and win1 is behind win2. Both windows are visible. The function `gtk_widget_set_visible (win1, true)` does nothing because win1 is already visible. So, win1 is still behind win2. The other function `gtk_window_present (win1)` moves win1 to the top of the stack of the windows. Therefore, if you want to present the window, you should use `gtk_window_present`.

Two functions `gtk_widget_show` and `gtk_widget_hide` is deprecated since GTK 4.10. You should use `gtk_widget_set_visible` instead.

Save the program as `pr3.c`, then compile and run it.

```
$ comp pr3
$ ./a.out
```

A small window appears.

Click on the close button then the window disappears and the program finishes.

3.2.2 GtkApplicationWindow

GtkApplicationWindow is a child object of GtkWindow. It has some extra feature for better integration with GtkApplication. It is recommended to use it as the top-level window of the application instead of GtkWindow.

Now rewrite the program and use GtkApplicationWindow.

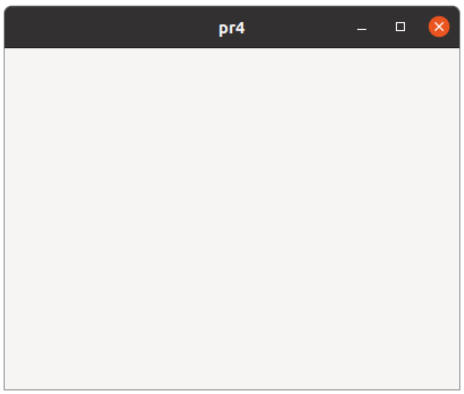


Figure 3: Screenshot of the window

```
1  static void
2  app_activate (GApplication *app, gpointer user_data) {
3    GtkWidget *win;
4
5    win = gtk_application_window_new (GTK_APPLICATION (app));
6    gtk_window_set_title (GTK_WINDOW (win), "pr4");
7    gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
8    gtk_window_present (GTK_WINDOW (win));
9  }
```

When you create GtkApplicationWindow, you need to give GtkApplication instance as an argument. Then it automatically connect these two instances. So you don't need to call `gtk_window_set_application` any more.

The program sets the title and the default size of the window. Compile it and run `a.out`, then you will see a bigger window with the title "pr4".

4 Widgets (1)

4.1 GtkLabel, GtkButton and GtkBox

4.1.1 GtkLabel

We made a window and showed it on the screen in the previous section. Now we go on to the next topic: widgets. The simplest widget is GtkLabel. It is a widget with text in it.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app) {
5    GtkWidget *win;
6    GtkWidget *lab;
7
8    win = gtk_application_window_new (GTK_APPLICATION (app));
9    gtk_window_set_title (GTK_WINDOW (win), "lb1");
10   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
11
12   lab = gtk_label_new ("Hello.");
13   gtk_window_set_child (GTK_WINDOW (win), lab);
14
15   gtk_window_present (GTK_WINDOW (win));
16 }
17
18 int
19 main (int argc, char **argv) {
20   GtkApplication *app;
```

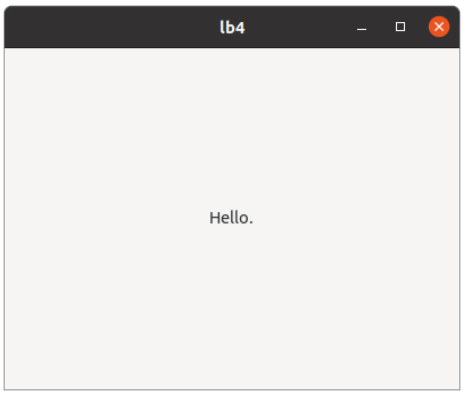


Figure 4: Screenshot of the label

```
21    int stat;
22
23    app = gtk_application_new ("com.github.ToshioCP.lb1", G_APPLICATION_DEFAULT_FLAGS);
24    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
25    stat =g_application_run (G_APPLICATION (app), argc, argv);
26    g_object_unref (app);
27    return stat;
28  }
```

Save this program to a file `lb1.c`. (You can use `src/misc/lb1.c` if you've downloaded this repository.) Then compile and run it.

```
$ comp lb1
$ ./a.out
```

A window with a message "Hello." appears.

There are only a few changes between `pr4.c` and `lb1.c`. A program `diff` is useful to know the difference.

```
$ cd misc; diff pr4.c lb1.c
4c4
< app_activate (GApplication *app, gpointer user_data) {
---
> app_activate (GApplication *app) {
5a6
>   GtkWidget *lab;
8c9
<   gtk_window_set_title (GTK_WINDOW (win), "pr4");
---
>   gtk_window_set_title (GTK_WINDOW (win), "lb1");
9a11,14
>
>   lab = gtk_label_new ("Hello.");
>   gtk_window_set_child (GTK_WINDOW (win), lab);
>
18c23
<   app = gtk_application_new ("com.github.ToshioCP.pr4",
    G_APPLICATION_DEFAULT_FLAGS);
---
>   app = gtk_application_new ("com.github.ToshioCP.lb1",
    G_APPLICATION_DEFAULT_FLAGS);
```

This tells us:

- A signal handler `app_activate` doesn't have `user_data` parameter. If the fourth argument of `g_signal_connect` is NULL, you can leave out `user_data`.
- The definition of a new variable `lab` is added.

- The title of the window is changed.
- A label is created and connected to the window as a child.

The function `gtk_window_set_child (GTK_WINDOW (win), lab)` makes the label `lab` a child widget of the window `win`. Be careful. A child widget is different from a child object. Objects have parent-child relationships and widgets also have parent-child relationships. But these two relationships are totally different. Don't be confused. In the program `lb1.c`, `lab` is a child widget of `win`. Child widgets are always located in their parent widget on the screen. See how the window has appeared on the screen. The application window includes the label.

The window `win` doesn't have any parents. We call such a window top-level window. An application can have more than one top-level windows.

4.1.2 GtkButton

The next widget is GtkButton. It displays a button on the screen with a label or icon on it. In this subsection, we will make a button with a label. When the button is clicked, it emits a "clicked" signal. The following program shows how to catch the signal and do something.

```
1  #include <gtk/gtk.h>
2
3  static void
4  click_cb (GtkButton *btn) {
5    g_print ("Clicked.\n");
6  }
7
8  static void
9  app_activate (GApplication *app) {
10   GtkWidget *win;
11   GtkWidget *btn;
12
13   win = gtk_application_window_new (GTK_APPLICATION (app));
14   gtk_window_set_title (GTK_WINDOW (win), "lb2");
15   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
16
17   btn = gtk_button_new_with_label ("Click␣me");
18   gtk_window_set_child (GTK_WINDOW (win), btn);
19   g_signal_connect (btn, "clicked", G_CALLBACK (click_cb), NULL);
20
21   gtk_window_present (GTK_WINDOW (win));
22 }
23
24 int
25 main (int argc, char **argv) {
26   GtkApplication *app;
27   int stat;
28
29   app = gtk_application_new ("com.github.ToshioCP.lb2", G_APPLICATION_DEFAULT_FLAGS);
30   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
31   stat =g_application_run (G_APPLICATION (app), argc, argv);
32   g_object_unref (app);
33   return stat;
34 }
```

Look at the line 17 to 19. First, it creates a GtkButton instance `btn` with a label "Click me". Then, adds the button to the window `win` as a child. Finally, connects the "clicked" signal of the button to the handler `click_cb`. So, if `btn` is clicked, the function `click_cb` is invoked. The suffix "cb" means "call back".

Name the program `lb2.c` and save it. Now compile and run it.

A window with the button appears. Click the button (it is a large button, you can click everywhere in the window), then a string "Clicked." appears on the terminal. It shows the handler was invoked by clicking the button.

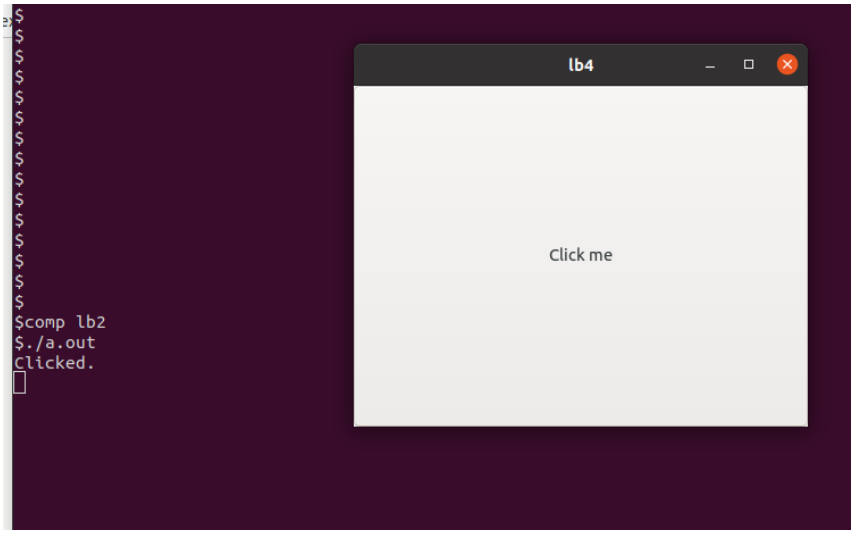


Figure 5: Screenshot of the label

It's good that we've made sure that the clicked signal was caught and the handler was invoked by using `g_print`. However, using `g_print` is out of harmony with GTK, which is a GUI library. So, we will change the handler. The following code is extracted from `lb3.c`.

```
1  static void
2  click_cb (GtkButton *btn, GtkWindow *win) {
3    gtk_window_destroy (win);
4  }
5
6  static void
7  app_activate (GApplication *app) {
8    GtkWidget *win;
9    GtkWidget *btn;
10
11   win = gtk_application_window_new (GTK_APPLICATION (app));
12   gtk_window_set_title (GTK_WINDOW (win), "lb3");
13   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
14
15   btn = gtk_button_new_with_label ("Close");
16   gtk_window_set_child (GTK_WINDOW (win), btn);
17   g_signal_connect (btn, "clicked", G_CALLBACK (click_cb), win);
18
19   gtk_window_present (GTK_WINDOW (win));
20 }
```

And the difference between `lb2.c` and `lb3.c` is as follows.

```
$ cd misc; diff lb2.c lb3.c
4,5c4,5
< click_cb (GtkButton *btn) {
<   g_print ("Clicked.\n");
---
> click_cb (GtkButton *btn, GtkWindow *win) {
>   gtk_window_destroy (win);
14c14
<   gtk_window_set_title (GTK_WINDOW (win), "lb2");
---
>   gtk_window_set_title (GTK_WINDOW (win), "lb3");
17c17
<   btn = gtk_button_new_with_label ("Click me");
---
>   btn = gtk_button_new_with_label ("Close");
19c19
```

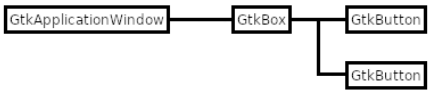


Figure 6: Parent-child relationship

```
<   g_signal_connect (btn, "clicked", G_CALLBACK (click_cb), NULL);
---
>   g_signal_connect (btn, "clicked", G_CALLBACK (click_cb), win);
29c29
<   app = gtk_application_new ("com.github.ToshioCP.lb2",
    G_APPLICATION_DEFAULT_FLAGS);
---
>   app = gtk_application_new ("com.github.ToshioCP.lb3",
    G_APPLICATION_DEFAULT_FLAGS);
35d34
<
```

The changes are:

- The function `g_print` in `lb2.c` was deleted and two lines are inserted.
  - `click_cb` has the second parameter, which comes from the fourth argument of the `g_signal_connect` at line 17. One thing to be careful is the types are different between the second parameter of `click_cb` and the fourth argument of `g_signal_connect`. The former is `GtkWindow *` and the latter is `GtkWidget *`. The compiler doesn't complain because `g_signal_connect` uses gpointer (general type of pointer). In this program the instance pointed by `win` is a GtkApplicationWindow object. It is a descendant of GtkWindow and GtkWidget class, so both `GtkWindow *` and `GtkWidget *` are correct types of the instance.
  - `gtk_destroy (win)` destroys the top-level window. Then the application quits.
- The label of `btn` is changed from "Click me" to "Close".
- The fourth argument of `g_signal_connect` is changed from `NULL` to `win`.

The most important change is the fourth argument of the `g_signal_connect`. This argument is described as "data to pass to the handler" in the definition of `g_signal_connect`.

4.1.3 GtkBox

GtkWindow and GtkApplicationWindow can have only one child. If you want to add two or more widgets in a window, you need a container widget. GtkBox is one of the containers. It arranges two or more child widgets into a single row or column. The following procedure shows the way to add two buttons in a window.

- Create a GtkApplicationWindow instance.
- Create a GtkBox instance and add it to the GtkApplicationWindow as a child.
- Create a GtkButton instance and append it to the GtkBox.
- Create another GtkButton instance and append it to the GtkBox.

After this, the Widgets are connected as the following diagram.

The program `lb4.c` is as follows.

```
1  #include <gtk/gtk.h>
2
3  static void
4  click1_cb (GtkButton *btn) {
5    const char *s;
6
7    s = gtk_button_get_label (btn);
8    if (g_strcmp0 (s, "Hello.") == 0)
9      gtk_button_set_label (btn, "Good-bye.");
10   else
```

```
11      gtk_button_set_label (btn, "Hello.");
12  }
13
14  static void
15  click2_cb (GtkButton *btn, GtkWindow *win) {
16    gtk_window_destroy (win);
17  }
18
19  static void
20  app_activate (GApplication *app) {
21    GtkWidget *win;
22    GtkWidget *box;
23    GtkWidget *btn1;
24    GtkWidget *btn2;
25
26    win = gtk_application_window_new (GTK_APPLICATION (app));
27    gtk_window_set_title (GTK_WINDOW (win), "lb4");
28    gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
29
30    box = gtk_box_new (GTK_ORIENTATION_VERTICAL, 5);
31    gtk_box_set_homogeneous (GTK_BOX (box), TRUE);
32    gtk_window_set_child (GTK_WINDOW (win), box);
33
34    btn1 = gtk_button_new_with_label ("Hello.");
35    g_signal_connect (btn1, "clicked", G_CALLBACK (click1_cb), NULL);
36
37    btn2 = gtk_button_new_with_label ("Close");
38    g_signal_connect (btn2, "clicked", G_CALLBACK (click2_cb), win);
39
40    gtk_box_append (GTK_BOX (box), btn1);
41    gtk_box_append (GTK_BOX (box), btn2);
42
43    gtk_window_present (GTK_WINDOW (win));
44  }
45
46  int
47  main (int argc, char **argv) {
48    GtkApplication *app;
49    int stat;
50
51    app = gtk_application_new ("com.github.ToshioCP.lb4", G_APPLICATION_DEFAULT_FLAGS);
52    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
53    stat =g_application_run (G_APPLICATION (app), argc, argv);
54    g_object_unref (app);
55    return stat;
56  }
```

Look at the function `app_activate`.

After the creation of a GtkApplicationWindow instance, a GtkBox instance is created.

```
box = gtk_box_new(GTK_ORIENTATION_VERTICAL, 5);
gtk_box_set_homogeneous (GTK_BOX (box), TRUE);
```

The first argument arranges the children of the box vertically. The orientation constants are defined like this:

- GTK_ORIENTATION_VERTICAL: the children widgets are arranged vertically
- GTK_ORIENTATION_HORIZONTAL: the children widgets are arranged horizontally

The second argument is the size of the space between the children. The unit of the length is pixel.

The next function fills the box with the children, giving them the same space.

After that, two buttons `btn1` and `btn2` are created and the signal handlers are set. Then, these two buttons are appended to the box.

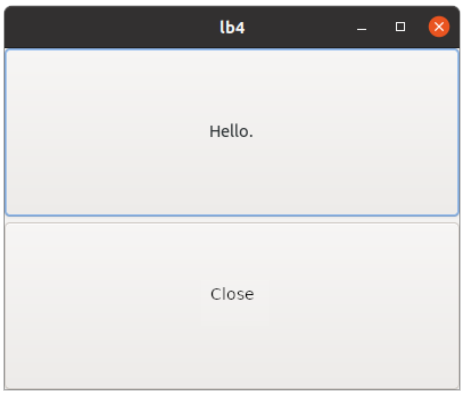


Figure 7: Screenshot of the box

```
1  static void
2  click1_cb (GtkButton *btn) {
3    const char *s;
4
5    s = gtk_button_get_label (btn);
6    if (g_strcmp0 (s, "Hello.") == 0)
7      gtk_button_set_label (btn, "Good-bye.");
8    else
9      gtk_button_set_label (btn, "Hello.");
10 }
```

The function `gtk_button_get_label` returns a text from the label. The string is owned by the button and you can't modify or free it. The `const` qualifier is necessary for the string `s`. If you change the string, your compiler will give you a waring.

You always need to be careful with the const qualifier when you see the GTK 4 API reference.

The handler corresponding to `btn1` toggles its label. The handler corresponding to `btn2` destroys the top-level window and the application quits.

5 Widgets (2)

5.1 GtkTextView, GtkTextBuffer and GtkScrolledWindow

5.1.1 GtkTextView and GtkTextBuffer

GtkTextView is a widget for multi-line text editing. GtkTextBuffer is a text buffer which is connected to GtkTextView. See the sample program `tfv1.c` below.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app) {
5    GtkWidget *win;
6    GtkWidget *tv;
7    GtkTextBuffer *tb;
8    gchar *text;
9
10   text =
11       "Once␣upon␣a␣time,␣there␣was␣an␣old␣man␣who␣was␣called␣Taketori-no-Okina.␣"
12       "It␣is␣a␣japanese␣word␣that␣means␣a␣man␣whose␣work␣is␣making␣bamboo␣baskets.\n"
13       "One␣day,␣he␣went␣into␣a␣hill␣and␣found␣a␣shining␣bamboo.␣"
14       "\"What␣a␣mysterious␣bamboo␣it␣is!,\"␣he␣said.␣"
15       "He␣cut␣it,␣then␣there␣was␣a␣small␣cute␣baby␣girl␣in␣it.␣"
16       "The␣girl␣was␣shining␣faintly.␣"
17       "He␣thought␣this␣baby␣girl␣is␣a␣gift␣from␣Heaven␣and␣took␣her␣home.\n"
18       "His␣wife␣was␣surprized␣at␣his␣story.␣"
```

```
19        "They␣were␣very␣happy␣because␣they␣had␣no␣children.␣"
20        ;
21    win = gtk_application_window_new (GTK_APPLICATION (app));
22    gtk_window_set_title (GTK_WINDOW (win), "Taketori");
23    gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
24
25    tv = gtk_text_view_new ();
26    tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
27    gtk_text_buffer_set_text (tb, text, -1);
28    gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
29
30    gtk_window_set_child (GTK_WINDOW (win), tv);
31
32    gtk_window_present (GTK_WINDOW (win));
33  }
34
35  int
36  main (int argc, char **argv) {
37    GtkApplication *app;
38    int stat;
39
40    app = gtk_application_new ("com.github.ToshioCP.tfv1",
          G_APPLICATION_DEFAULT_FLAGS);
41    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
42    stat = g_application_run (G_APPLICATION (app), argc, argv);
43    g_object_unref (app);
44    return stat;
45  }
```

Look at line 25. A GtkTextView instance is created and its pointer is assigned to `tv`. When the GtkTextView instance is created, a GtkTextBuffer instance is also created and connected to the GtkTextView automatically. "GtkTextBuffer instance" will be referred to simply as "GtkTextBuffer" or "buffer". In the next line, the pointer to the buffer is assigned to `tb`. Then, the text from line 10 to 20 is assigned to the buffer. If the third argument of `gtk_text_buffer_set_text` is a positive integer, it is the length of the text. It it is -1, the string terminates with NULL.

GtkTextView has a wrap mode. When it is set to `GTK_WRAP_WORD_CHAR`, text wraps in between words, or if that is not enough, also between graphemes.

Wrap mode is written in Gtk_WrapMode in the GTK 4 API document.

In line 30, `tv` is added to `win` as a child.

Now compile and run it. If you've downloaded this repository, its pathname is `src/tfv/tfv1.c`.

```
$ cd src/tfv
$ comp tfv1
$ ./a.out
```

There's an I-beam pointer in the window. You can add or delete any characters on the GtkTextView, and your changes are kept in the GtkTextBuffer. If you add more characters beyond the limit of the window, the height increases and the window extends. If the height gets bigger than the height of the screen, you won't be able to control the size of the window or change it back to the original size. This is a problem, that is to say a bug. This can be solved by adding a GtkScrolledWindow between the GtkApplicationWindow and GtkTextView.

5.1.2 GtkScrolledWindow

What we need to do is:

- Create a GtkScrolledWindow and insert it as a child of the GtkApplicationWindow
- Insert the GtkTextView widget to the GtkScrolledWindow as a child.

Modify `tfv1.c` and save it as `tfv2.c`. There is only a few difference between these two files.

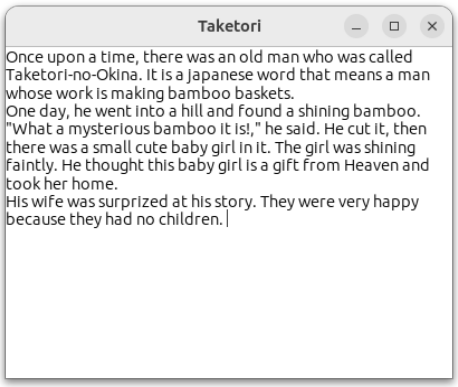


Figure 8: GtkTextView

```
$ cd tfv; diff tfv1.c tfv2.c
5a6
>   GtkWidget *scr;
24a26,28
>   scr = gtk_scrolled_window_new ();
>   gtk_window_set_child (GTK_WINDOW (win), scr);
>
30c34
<   gtk_window_set_child (GTK_WINDOW (win), tv);
---
>   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
40c44
<   app = gtk_application_new ("com.github.ToshioCP.tfv1",
    G_APPLICATION_DEFAULT_FLAGS);
---
>   app = gtk_application_new ("com.github.ToshioCP.tfv2",
    G_APPLICATION_DEFAULT_FLAGS);
```

The whole code of `tfv2.c` is as follows.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app) {
5    GtkWidget *win;
6    GtkWidget *scr;
7    GtkWidget *tv;
8    GtkTextBuffer *tb;
9    gchar *text;
10
11   text =
12       "Once␣upon␣a␣time,␣there␣was␣an␣old␣man␣who␣was␣called␣Taketori-no-Okina.␣"
13       "It␣is␣a␣japanese␣word␣that␣means␣a␣man␣whose␣work␣is␣making␣bamboo␣baskets.\n"
14       "One␣day,␣he␣went␣into␣a␣hill␣and␣found␣a␣shining␣bamboo.␣"
15       "\"What␣a␣mysterious␣bamboo␣it␣is!,\"␣he␣said.␣"
16       "He␣cut␣it,␣then␣there␣was␣a␣small␣cute␣baby␣girl␣in␣it.␣"
17       "The␣girl␣was␣shining␣faintly.␣"
18       "He␣thought␣this␣baby␣girl␣is␣a␣gift␣from␣Heaven␣and␣took␣her␣home.\n"
19       "His␣wife␣was␣surprized␣at␣his␣story.␣"
20       "They␣were␣very␣happy␣because␣they␣had␣no␣children.␣"
21       ;
22   win = gtk_application_window_new (GTK_APPLICATION (app));
23   gtk_window_set_title (GTK_WINDOW (win), "Taketori");
24   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
25
26   scr = gtk_scrolled_window_new ();
27   gtk_window_set_child (GTK_WINDOW (win), scr);
```

```
28
29    tv = gtk_text_view_new ();
30    tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
31    gtk_text_buffer_set_text (tb, text, -1);
32    gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
33
34    gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
35
36    gtk_window_present (GTK_WINDOW (win));
37  }
38
39  int
40  main (int argc, char **argv) {
41    GtkApplication *app;
42    int stat;
43
44    app = gtk_application_new ("com.github.ToshioCP.tfv2",
          G_APPLICATION_DEFAULT_FLAGS);
45    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
46    stat = g_application_run (G_APPLICATION (app), argc, argv);
47    g_object_unref (app);
48    return stat;
49  }
```

Compile and run it.

Now, the window doesn't extend even if you type a lot of characters, it just scrolls.

6 Strings and memory management

GtkTextView and GtkTextBuffer have functions that have string parameters or return a string. The knowledge of strings and memory management is useful to understand how to use these functions.

6.1 String and memory

A String is an array of characters that is terminated with '\0'. String is not a C type such as char, int, float or double, but a character array. It behaves like a string in other languages. So, the pointer is often called 'a string'.

The following is a sample program.

```
char a[10], *b;

a[0] = 'H';
a[1] = 'e';
a[2] = 'l';
a[3] = 'l';
a[4] = 'o';
a[5] = '\0';

b = a;
/* *b is 'H' */
/* *(++b) is 'e' */
```

An array `a` is defined as a `char` type array and its size is ten. The first five elements are 'H', 'e', 'l', 'l', 'o'. They are character codes. For example, 'H' is the same as 0x48 or 72. The sixth element is '\0', which is the same as zero, and indicates that the sequence of the data ends there. The array represents the string "Hello".

The size of the array is 10, so four bytes aren't used. But it's OK. They are just ignored. (If the variable `a` is defined out of functions or its class is static, the four bytes are assigned with zero. Otherwise, that is to say, the class is auto or register, they are undefined.)

The variable `b` is a pointer to a character. It is assigned with `a`, so `b` points the first element of `a` (character 'H'). The array `a` is immutable. So `a=a+1` causes syntax error.

On the other hand, `b` is a pointer type variable, which is mutable. So, `++b`, which increases `b` by one, is allowed.

If a pointer is NULL, it points nothing. So, the pointer is not a string. It is different from empty string. Empty string is a pointer points '\0'.

There are four cases:

- The string is read only
- The string is in static memory area
- The string is in stack
- The string is in memory allocated from the heap area

6.2 Read only string

A string literal is surrounded by double quotes like this:

```
char *s;
s = "Hello"
```

"Hello" is a string literal, and is read only. So, the following program is illegal.

```
*(s+1) = 'a';
```

The result is undefined. Probably a bad thing will happen, for example, a segmentation fault.

NOTE: The memory of the literal string is allocated when the program is compiled. It is possible to see the literal strings with `strings` command.

```
$ strings src/tvf/a.out
/lib64/ld-linux-x86-64.so.2
cN<5
... ... ...
... ... ...
Once upon a time, there was an old man who was called Taketori-no-Okina. It is a
    japanese word that means a man whose work is making bamboo baskets.
One day, he went into a hill and found a shining bamboo. "What a mysterious bamboo
    it is!," he said. He cut it, then there was a small cute baby girl in it. The
    girl was shining faintly. He thought this baby girl is a gift from Heaven and
    took her home.
His wife was surprized at his story. They were very happy because they had no
    children.
... ... ...
... ... ...
```

It tells us that literal strings are embedded in program binary codes.

6.3 Strings defined as arrays

If a string is defined as an array, it's stored in static memory area or stack. It depends on the class of the array. If the array's class is `static`, then it's placed in static memory area. The allocated memory lives for the life of the program. This area is writable.

If the array's class is `auto`, it's placed in stack. If the array is defined inside a function, its default class is `auto`. The stack area will disappear when the function returns to the caller. Arrays defined on the stack are writable.

```
static char a[] = {'H', 'e', 'l', 'l', 'o', '\0'};

void
print_strings (void) {
  char b[] = "Hello";
```

```
  a[1] = 'a'; /* Because the array is static, it's writable. */
  b[1] = 'a'; /* Because the array is auto, it's writable. */

  printf ("%s\n", a); /* Hallo */
  printf ("%s\n", b); /* Hallo */
}
```

The array `a` is defined out of functions. It is placed in the static memory area even if the `static` class is left out. The compiler calculates the number of the elements (six) and allocates six bytes in the static memory area. Then, it copies "Hello" literal string data to the memory.

The array `b` is defined inside the function, so its class is `auto`. The compiler calculates the number of the elements in the string literal. It is six because it has '\0' terminator. The compiler allocates six bytes in the stack and copies "Hello" litaral string to the stack memory.

Both `a` and `b` are writable.

The memory is allocated and freed by the program automatically so you don't need to allocate or free. The array `a` is alive during the program's life time. The array `b` is alive when the function is called until the function returns to the caller.

6.4 Strings in the heap area

You can get, use and release memory from the heap area. The standard C library provides `malloc` to get memory and `free` to put back memory. GLib provides the functions `g_new` and `g_free`. They are similar to `malloc` and `free`.

```
g_new (struct_type, n_struct)
```

`g_new` is a macro to allocate memory for an array.

- `struct_type` is the type of the element of the array.
- `n_struct` is the size of the array.
- The return value is a pointer to the array. Its type is a pointer to `struct_type`.

For example,

```
char *s;
s = g_new (char, 10);
/* s points an array of char. The size of the array is 10. */

struct tuple {int x, y;} *t;
t = g_new (struct tuple, 5);
/* t points an array of struct tuple. */
/* The size of the array is 5. */
```

`g_free` frees memory.

```
void
g_free (gpointer mem);
```

If `mem` is NULL, `g_free` does nothing. `gpointer` is a type of general pointer. It is the same as `void *`. This pointer can be casted to any pointer type. Conversely, any pointer type can be casted to `gpointer`.

```
g_free (s);
/* Frees the memory allocated to s. */

g_free (t);
/* Frees the memory allocated to t. */
```

If the argument doesn't point allocated memory it will cause an error, specifically, a segmentation fault.

Some GLib functions allocate memory. For example, `g_strdup` allocates memory and copies a string given as an argument.

```
char *s;
s = g_strdup ("Hello");
g_free (s);
```

The string literal "Hello" has 6 bytes because the string has '\0' at the end. `g_strdup` gets 6 bytes from the heap area and copies the string to the memory. `s` is assigned the start address of the memory. `g_free` returns the memory to the heap area.

`g_strdup` is described in GLib API Reference. The following is extracted from the reference.

> The returned string should be freed with `g_free()` when no longer needed.

If you forget to free the allocated memory it will remain until the program ends. Repeated allocation and no freeing cause memory leak. It is a bug and may bring a serious problem.

6.5 const qualifier

A `const` qualified variable can be assigned to initialize it. Once it is initialized, it is never allowed to change or free.

```
const int x = 10; /* initialization is OK. */

x = 20; /* This is illegal because x is qualified with const */
```

If a function returns `const char*` type, the string can't be changed or freed. If a function has a `const char *` type parameter, it ensures that the parameter is not changed in the function.

```
// You never change or free the returned string.
const char*
gtk_label_get_text (
  GtkLabel* self
)

// Str keeps itself during the function runs
void
gtk_label_set_text (
  GtkLabel* self,
  const char* str
)
```

7 Widgets (3)

7.1 Open signal

7.1.1 G_APPLICATION_HANDLES_OPEN flag

We made a very simple editor in the previous section with GtkTextView, GtkTextBuffer and GtkScrolledWindow. We will add file-read ability to the program and improve it to a file viewer.

The easiest way to give a filename is to use a command line argument.

```
$ ./a.out filename
```

The program will open the file and insert its contents into the GtkTextBuffer.

To do this, we need to know how GtkApplication (or GApplication) recognizes arguments. This is described in the GIO API Reference – Application.

When GtkApplication is created, a flag (GApplicationFlags) is given as an argument.

```
GtkApplication *
gtk_application_new (const gchar *application_id, GApplicationFlags flags);
```

This tutorial explains only two flags, `G_APPLICATION_DEFAULT_FLAGS` and `G_APPLICATION_HANDLES_OPEN`.

`G_APPLICATION_FLAGS_NONE` was used instead of `G_APPLICATION_DEFAULT_FLAGS` before GIO version2.73.3 (GLib 2.73.3 5/Aug/2022). Now it is deprecated and `G_APPLICATION_DEFAULT_FLAGS` is recommended.

For further information, see GIO API Reference – ApplicationFlags and GIO API Reference – g_application_run.

We've already used `G_APPLICATION_DEFAULT_FLAGS`, as it is the simplest option, and no command line arguments are allowed. If you give arguments, an error will occur.

The flag `G_APPLICATION_HANDLES_OPEN` is the second simplest option. It allows arguments but only filenames.

```
app = gtk_application_new ("com.github.ToshioCP.tfv3", G_APPLICATION_HANDLES_OPEN);
```

7.1.2 open signal

When `G_APPLICATION_HANDLES_OPEN` flag is given to the application, two signals are available.

- activate signal: This signal is emitted when there's no argument.
- open signal: This signal is emitted when there is at least one argument.

The handler of the "open" signal is defined as follows.

```
void
open (
  GApplication* self,
  gpointer files,
  gint n_files,
  gchar* hint,
  gpointer user_data
)
```

The parameters are:

- self: the application instance (usually GtkApplication)
- files: an array of GFiles. [array length=n_files] [element-type GFile]
- n_files: the number of the elements of `files`
- hint: a hint provided by the calling instance (usually it can be ignored)
- user_data: user data that is set when the signal handler was connected.

7.2 File viewer

7.2.1 What is a file viewer?

A file viewer is a program that displays text files. Our file viewer is as follows.

- When arguments are given, it recognizes the first argument as a filename and opens it.
- The second argument and after are ignored.
- If there's no argument, it shows an error message and quit.
- If it successfully opens the file, it reads the contents of the file, inserts them to GtkTextBuffer and shows the window.
- If it fails to open the file, it shows an error message and quit.

The program is shown below.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app) {
5    g_printerr ("You␣need␣a␣filename␣argument.\n");
6  }
7
8  static void
9  app_open (GApplication *app, GFile ** files, int n_files, char *hint) {
10   GtkWidget *win;
11   GtkWidget *scr;
12   GtkWidget *tv;
13   GtkTextBuffer *tb;
```

```
14    char *contents;
15    gsize length;
16    char *filename;
17    GError *err = NULL;
18
19    win = gtk_application_window_new (GTK_APPLICATION (app));
20    gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
21
22    scr = gtk_scrolled_window_new ();
23    gtk_window_set_child (GTK_WINDOW (win), scr);
24
25    tv = gtk_text_view_new ();
26    tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
27    gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
28    gtk_text_view_set_editable (GTK_TEXT_VIEW (tv), FALSE);
29    gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
30
31    if (g_file_load_contents (files[0], NULL, &contents, &length, NULL, &err)) {
32      gtk_text_buffer_set_text (tb, contents, length);
33      g_free (contents);
34      if ((filename = g_file_get_basename (files[0])) != NULL) {
35        gtk_window_set_title (GTK_WINDOW (win), filename);
36        g_free (filename);
37      }
38      gtk_window_present (GTK_WINDOW (win));
39    } else {
40      g_printerr ("%s.\n", err->message);
41      g_error_free (err);
42      gtk_window_destroy (GTK_WINDOW (win));
43    }
44  }
45
46  int
47  main (int argc, char **argv) {
48    GtkApplication *app;
49    int stat;
50
51    app = gtk_application_new ("com.github.ToshioCP.tfv3", G_APPLICATION_HANDLES_OPEN);
52    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
53    g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
54    stat = g_application_run (G_APPLICATION (app), argc, argv);
55    g_object_unref (app);
56    return stat;
57  }
```

Save it as `tfv3.c`. If you've downloaded this repository, the file is `src/tfv/tfv3.c`. Compile and run it.

```
$ comp tfv3
$ ./a.out tfv3.c
```

The function `main` has only two changes from the previous version.

- `G_APPLICATION_DEFAULT_FLAGS` is replaced by `G_APPLICATION_HANDLES_OPEN`
- `g_signal_connect (app, "open", G_CALLBACK (app_open), NULL)` is added.

When the flag `G_APPLICATION_HANDLES_OPEN` is given to `gtk_application_new` function, the application behaves like this:

- If the application is run without command line arguments, it emits "activate" signal when it is activated.
- If the application is run with command line arguments, it emits "open" signal when it is activated.

The handler `app_activate` becomes very simple. It just outputs an error message and returns to the caller. Then the application quits immediately because no window is created.

The main work is done in the handler `app_open`.

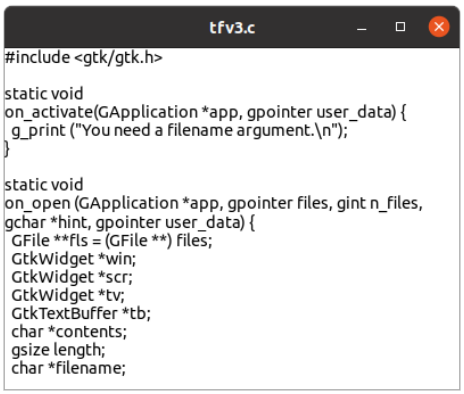


Figure 9: File viewer

- Creates GtkApplicationWindow, GtkScrolledWindow, GtkTextView and GtkTextBuffer and connects them together
- Sets wrap mode to `GTK_WRAP_WORD_CHAR` in GtktextView
- Sets GtkTextView to non-editable because the program isn't an editor but only a viewer
- Reads the file and inserts the text into GtkTextBuffer (this will be explained later)
- If the file is not opened, outputs an error message and destroys the window. This makes the application quit.

The following is the file reading part of the program.

```
if (g_file_load_contents (files[0], NULL, &contents, &length, NULL, &err)) {
  gtk_text_buffer_set_text (tb, contents, length);
  g_free (contents);
  if ((filename = g_file_get_basename (files[0])) != NULL) {
    gtk_window_set_title (GTK_WINDOW (win), filename);
    g_free (filename);
  }
  gtk_window_present (GTK_WINDOW (win));
} else {
  g_printerr ("%s.\n", err->message);
  g_error_free (err);
  gtk_window_destroy (GTK_WINDOW (win));
}
```

The function `g_file_load_contents` loads the file contents into a temporary buffer, which is automatically allocated and sets `contents` to point the buffer. The length of the buffer is assigned to `length`. It returns `TRUE` if the file's contents are successfully loaded. If an error occurs, it returns `FALSE` and sets the variable `err` to point a newly created GError structure. The caller takes ownership of the GError structure and is responsible for freeing it. If you want to know the details about g_file_load_contents, see g file load contents.

If it has successfully read the file, it inserts the contents into GtkTextBuffer, frees the temporary buffer pointed by `contents`, sets the title of the window, frees the memories pointed by `filename` and then shows the window.

If it fails, `g_file_load_contents` sets `err` to point a newly created GError structure. The structure is:

```
struct GError {
  GQuark domain;
  int code;
  char* message;
}
```

The `message` member is used most often. It points an error message. A function `g_error_free` is used to free the memory of the structure. See GError.

The program above outputs an error message, frees `err` and destroys the window and finally make the program quit.

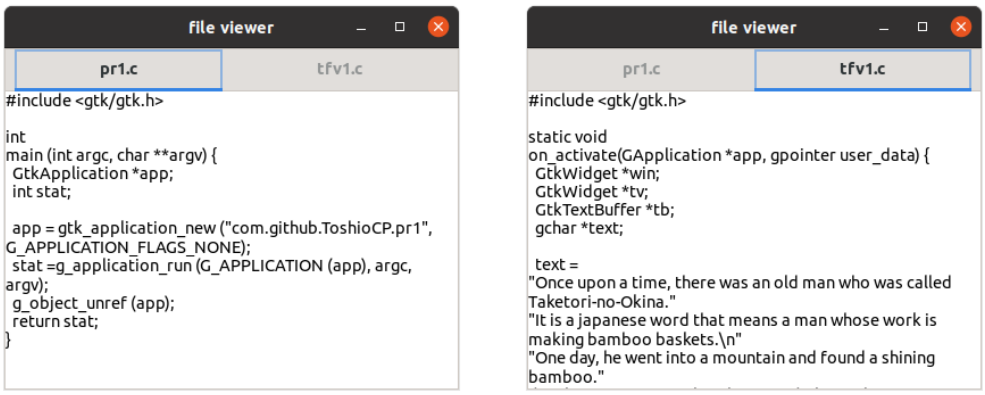


Figure 10: GtkNotebook

7.3 GtkNotebook

GtkNotebook is a container widget that contains multiple widgets with tabs. It shows only one child at a time. Another child will be shown when its tab is clicked.

The left image is the window at the startup. The file `pr1.c` is shown and its filename is in the left tab. After clicking on the right tab, the contents of the file `tfv1.c` is shown (the right image).

The following is `tfv4.c`. It has GtkNoteBook widget. It is inserted as a child of GtkApplicationWindow and contains multiple GtkScrolledWindow.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *app) {
5    g_printerr ("You␣need␣filename␣arguments.\n");
6  }
7
8  static void
9  app_open (GApplication *app, GFile ** files, gint n_files, gchar *hint) {
10   GtkWidget *win;
11   GtkWidget *nb;
12   GtkWidget *lab;
13   GtkNotebookPage *nbp;
14   GtkWidget *scr;
15   GtkWidget *tv;
16   GtkTextBuffer *tb;
17   char *contents;
18   gsize length;
19   char *filename;
20   int i;
21   GError *err = NULL;
22
23   win = gtk_application_window_new (GTK_APPLICATION (app));
24   gtk_window_set_title (GTK_WINDOW (win), "file␣viewer");
25   gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
26   nb = gtk_notebook_new ();
27   gtk_window_set_child (GTK_WINDOW (win), nb);
28
29   for (i = 0; i < n_files; i++) {
30     if (g_file_load_contents (files[i], NULL, &contents, &length, NULL, &err)) {
31       scr = gtk_scrolled_window_new ();
32       tv = gtk_text_view_new ();
33       tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
34       gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
35       gtk_text_view_set_editable (GTK_TEXT_VIEW (tv), FALSE);
36       gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
37
```

```
38        gtk_text_buffer_set_text (tb, contents, length);
39        g_free (contents);
40        if ((filename = g_file_get_basename (files[i])) != NULL) {
41          lab = gtk_label_new (filename);
42          g_free (filename);
43        } else
44          lab = gtk_label_new ("");
45        gtk_notebook_append_page (GTK_NOTEBOOK (nb), scr, lab);
46        nbp = gtk_notebook_get_page (GTK_NOTEBOOK (nb), scr);
47        g_object_set (nbp, "tab-expand", TRUE, NULL);
48      } else {
49        g_printerr ("%s.\n", err->message);
50        g_clear_error (&err);
51      }
52    }
53    if (gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb)) > 0)
54      gtk_window_present (GTK_WINDOW (win));
55    else
56      gtk_window_destroy (GTK_WINDOW (win));
57  }
58
59  int
60  main (int argc, char **argv) {
61    GtkApplication *app;
62    int stat;
63
64    app = gtk_application_new ("com.github.ToshioCP.tfv4", G_APPLICATION_HANDLES_OPEN);
65    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
66    g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
67    stat = g_application_run (G_APPLICATION (app), argc, argv);
68    g_object_unref (app);
69    return stat;
70  }
```

Most of the changes are in the function `app_open`. The numbers at the left of the following items are line numbers in the source code.

- 11-13: Variables `nb`, `lab` and `nbp` are defined. They point GtkNotebook, GtkLabel and GtkNotebookPage respectively.
- 24: The window's title is set to "file viewer".
- 25: The default size of the window is 600x400.
- 26-27: GtkNotebook is created and inserted to the GtkApplicationWindow as a child.
- 29-57: For-loop. The variable `files[i]` points i-th GFile, which is created by the GtkApplication from the i-th command line argument.
- 31-36: GtkScrollledWindow, GtkTextView are created. GtkTextBuffer is got from the GtkTextView. The GtkTextView is connected to the GtkScrolledWindow as a child.
- 38-39: inserts the contents of the file into GtkTextBuffer and frees the memory pointed by `contents`.
- 40-42: If the filename is taken from the GFile, GtkLabel is created with the filename. The string `filename` is freed..
- 43-44: If it fails to take the filename, empty string GtkLabel is created.
- 45-46: Appends a GtkScrolledWindow to the GtkNotebook as a child. And the GtkLabel is set as the child's tab. At the same time, a GtkNoteBookPage is created automatically. The function `gtk_notebook_get_page` returns the GtkNotebookPage of the child (GtkScrolledWindow).
- 47: GtkNotebookPage has "tab-expand" property. If it is set to TRUE then the tab expands horizontally as long as possible. If it is FALSE, then the width of the tab is determined by the size of the label. `g_object_set` is a general function to set properties of objects. See GObject API Reference – g_object_set.
- 48-50: If it fails to read the file, the error message is shown. The function `g_clear_error (&err)` works like `g_error_free (err); err = NULL`.
- 53-56: If at least one page exists, the window is shown. Otherwise, the window is destroyed and the application quits.

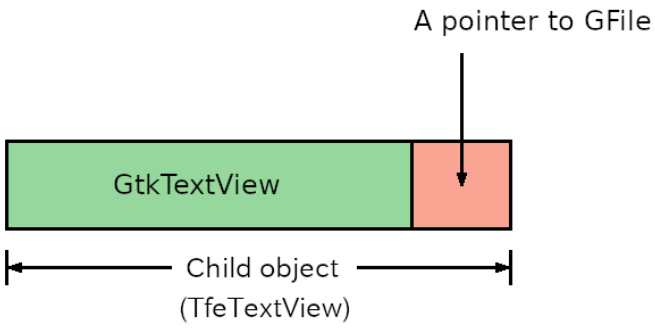


Figure 11: Child object of GtkTextView

8 Defining a final class

8.1 A very simple editor

We made a very simple file viewer in the previous section. Now we go on to rewrite it and turn it into very simple editor. Its source file is `tfe1.c` (text file editor 1) under `tfe` directory.

GtkTextView is a multi-line editor. So, we don't need to write the editor from scratch. We just add two things to the file viewer:

- Pointers to GFile instances.
- A text-save function.

There are a couple of ways to store the pointers.

- Use global variables
- Make a child class of GtkTextView and its each instance holds a pointer to the GFile instance.

Using global variables is easy to implement. Define a sufficient size pointer array to GFile. For example,

```
GFile *f[20];
```

The variable `f[i]` corresponds to the file associated with the i-th GtkNotebookPage.

However, There are two problems. The first is the size of the array. If a user gives too many arguments (more than 20 in the example above), it is impossible to store all the pointers to the GFile instances. The second is difficulty to maintain the program. We have a small program so far. But, the more you develop the program, the bigger its size grows. Generally speaking, it is very difficult to maintain global variables in a big program. When you check the global variable, you need to check all the codes that use the variable.

Making a child class is a good idea in terms of maintenance. And we prefer it rather than a global variable.

Be careful that we are thinking about "child class", not "child widget". Child class and child widget are totally different. Class is a term of GObject system. If you are not familiar with GObject, see:

- GObject API reference
- GObject tutorial for beginners

A child class inherits everything from the parent and, in addition, extends its performance. We will define TfeTextView as a child class of GtkTextView. It has everything that GtkTextView has and adds a pointer to a GFile.

8.2 How to define a child class of GtkTextView

You need to know GObject system convention. First, look at the program below.

```
#define TFE_TYPE_TEXT_VIEW tfe_text_view_get_type ()
G_DECLARE_FINAL_TYPE (TfeTextView, tfe_text_view, TFE, TEXT_VIEW, GtkTextView)

struct _TfeTextView
{
```

```
  GtkTextView parent;
  GFile *file;
};

G_DEFINE_FINAL_TYPE (TfeTextView, tfe_text_view, GTK_TYPE_TEXT_VIEW);

static void
tfe_text_view_init (TfeTextView *tv) {
}

static void
tfe_text_view_class_init (TfeTextViewClass *class) {
}

void
tfe_text_view_set_file (TfeTextView *tv, GFile *f) {
  tv -> file = f;
}

GFile *
tfe_text_view_get_file (TfeTextView *tv) {
  return tv -> file;
}

GtkWidget *
tfe_text_view_new (void) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_TEXT_VIEW, NULL));
}
```

- TfeTextView is divided into two parts. Tfe and TextView. Tfe is called prefix or namespace. TextView is called object.
- There are three different identifier patterns. TfeTextView (camel case), tfe_text_view (this is used for functions) and TFE_TEXT_VIEW (This is used to cast a object to TfeTextView).
- First, define `TFE_TYPE_TEXT_VIEW` macro as `tfe_text_view_get_type ()`. The name is always (prefix)_TYPE_(object) and the letters are upper case. And the replacement text is always (prefix)_(object)_get_type () and the letters are lower case. This definition is put before `G_DECLARE_FINAL_TYPE` macro.
- The arguments of `G_DECLARE_FINAL_TYPE` macro are the child class name in camel case, lower case with underscore, prefix (upper case), object (upper case with underscore) and parent class name (camel case). The following two C structures are declared in the expansion of the macro.
  - `typedef struct _TfeTextView TfeTextView`
  - `typedef struct {GtkTextViewClass parent_class; } TfeTextViewClass;`
- These declaration tells us that TfeTextView and TfeTextViewClass are C structures. "TfeTextView" has two meanings, class name and C structure name. The C structure TfeTextView is called object. Similarly, TfeTextViewClass is called class.
- Declare the structure `_TfeTextView`. The underscore is necessary. The first member is the parent object (C structure). Notice this is not a pointer but the object itself. The second member and after are members of the child object. TfeTextView structure has a pointer to a GFile instance as a member.
- `G_DEFINE_FINEL_TYPE` macro. The arguments are the child object name in camel case, lower case with underscore and parent object type (prefix)_TYPE_(module). This macro is mainly used to register the new class to the type system. Type system is a base system of GObject. Every class has its own type. The types of GObject, GtkWidget and TfeTextView are `G_TYPE_OBJECT`, `GTK_TYPE_WIDGET` and `TFE_TYPE_TEXT_VIEW` respectively. For example, `TFE_TYPE_TEXT_VIEW` is a macro and it is expanded to a function `tfe_text_view_get_type()`. It returns a integer which is unique among all GObject system classes.
- The instance init function `tfe_text_view_init` is called when the instance is created. It is the same as a constructor in other object oriented languages.
- The class init function `tfe_text_view_class_init` is called when the class is created.
- Two functions `tfe_text_view_set_file` and `tfe_text_view_get_file` are public functions. Public functions are open and you can call them anywhere. They are the same as public method in other

object oriented languages. `tv` is a pointer to the TfeTextView object (C structure). It has a member `file` and it is pointed by `tv->file`.

- TfeTextView instance creation function is `tfe_text_view_new`. Its name is (prefix)_(object)_new. It uses `g_object_new` function to create the instance. The arguments are (prefix)_TYPE_(object), a list to initialize properties and NULL. NULL is the end mark of the property list. No property is initialized here. And the return value is casted to GtkWidget.

This program shows the outline how to define a child class.

8.3 Close-request signal

Imagine that you are using this editor. First, you run the editor with arguments. The arguments are filenames. The editor reads the files and shows the window with the text of files in it. Then you edit the text. After you finish editing, you click on the close button of the window and quit the editor. The editor updates files just before the window closes.

GtkWindow emits the "close-request" signal when the close button is clicked. We will connect the signal and the handler `before_close`. (A handler is a C function which is connected to a signal.) The function `before_close` is called when the signal "close-request" is emitted.

```
g_signal_connect (win, "close-request", G_CALLBACK (before_close), NULL);
```

The argument `win` is a GtkApplicationWindow, in which the signal "close-request" is defined, and `before_close` is the handler. The `G_CALLBACK` cast is necessary for the handler. The program of `before_close` is as follows.

```
1  static gboolean
2  before_close (GtkWindow *win, GtkWidget *nb) {
3    GtkWidget *scr;
4    GtkWidget *tv;
5    GFile *file;
6    char *pathname;
7    GtkTextBuffer *tb;
8    GtkTextIter start_iter;
9    GtkTextIter end_iter;
10   char *contents;
11   unsigned int n;
12   unsigned int i;
13   GError *err = NULL;
14
15   n = gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb));
16   for (i = 0; i < n; ++i) {
17     scr = gtk_notebook_get_nth_page (GTK_NOTEBOOK (nb), i);
18     tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
19     file = tfe_text_view_get_file (TFE_TEXT_VIEW (tv));
20     tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
21     gtk_text_buffer_get_bounds (tb, &start_iter, &end_iter);
22     contents = gtk_text_buffer_get_text (tb, &start_iter, &end_iter, FALSE);
23     if (! g_file_replace_contents (file, contents, strlen (contents), NULL, TRUE,
           G_FILE_CREATE_NONE, NULL, NULL, &err)) {
24       g_printerr ("%s.\n", err->message);
25       g_clear_error (&err);
26     }
27     g_free (contents);
28     g_object_unref (file);
29   }
30   return FALSE;
31 }
```

The numbers on the left are line numbers.

- 15: The number of note book pages is assigned to `n`.
- 16-29: For loop with regard to the index to each page.
- 17-19: `scr`, `tv` and `file` is assigned pointers to the GtkScrolledWindow, TfeTextView and GFile. The GFile of TfeTextView was stored when `app_open` handler was called. It will be shown later.

- 20-22: `tb` is assigned the GtkTextBuffer of the TfeTextView. The contents of the buffer are accessed with iterators. Iterators points somewhere in the buffer. The function `gtk_text_buffer_get_bounds` assigns the start and end of the buffer to `start_iter` and `end_iter` respectively. Then the function `gtk_text_buffer_get_text` returns the text between `start_iter` and `end_iter`, which is the whole text in the buffer.
- 23-26: The text is saved to the file. If it fails, error messages are displayed. The GError instance must be freed and the pointer `err` needs to be NULL for the next run in the loop.
- 27: `contents` are freed.
- 28: GFile is useless. `g_object_unref` decreases the reference count of the GFile. Reference count will be explained in the later section. The reference count will be zero and the GFile instance will destroy itself.

8.4 Source code of tfe1.c

The following is the whole source code of `tfe1.c`.

```
1  #include <gtk/gtk.h>
2
3  /* Define TfeTextView Widget which is the child class of GtkTextView */
4
5  #define TFE_TYPE_TEXT_VIEW tfe_text_view_get_type ()
6  G_DECLARE_FINAL_TYPE (TfeTextView, tfe_text_view, TFE, TEXT_VIEW, GtkTextView)
7
8  struct _TfeTextView
9  {
10   GtkTextView parent;
11   GFile *file;
12 };
13
14 G_DEFINE_FINAL_TYPE (TfeTextView, tfe_text_view, GTK_TYPE_TEXT_VIEW);
15
16 static void
17 tfe_text_view_init (TfeTextView *tv) {
18   tv->file = NULL;
19 }
20
21 static void
22 tfe_text_view_class_init (TfeTextViewClass *class) {
23 }
24
25 void
26 tfe_text_view_set_file (TfeTextView *tv, GFile *f) {
27   tv->file = f;
28 }
29
30 GFile *
31 tfe_text_view_get_file (TfeTextView *tv) {
32   return tv -> file;
33 }
34
35 GtkWidget *
36 tfe_text_view_new (void) {
37   return GTK_WIDGET (g_object_new (TFE_TYPE_TEXT_VIEW, NULL));
38 }
39
40 /* ---------- end of the definition of TfeTextView ---------- */
41
42 static gboolean
43 before_close (GtkWindow *win, GtkWidget *nb) {
44   GtkWidget *scr;
45   GtkWidget *tv;
46   GFile *file;
47   char *pathname;
48   GtkTextBuffer *tb;
```

```
49    GtkTextIter start_iter;
50    GtkTextIter end_iter;
51    char *contents;
52    unsigned int n;
53    unsigned int i;
54    GError *err = NULL;
55
56    n = gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb));
57    for (i = 0; i < n; ++i) {
58      scr = gtk_notebook_get_nth_page (GTK_NOTEBOOK (nb), i);
59      tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
60      file = tfe_text_view_get_file (TFE_TEXT_VIEW (tv));
61      tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
62      gtk_text_buffer_get_bounds (tb, &start_iter, &end_iter);
63      contents = gtk_text_buffer_get_text (tb, &start_iter, &end_iter, FALSE);
64      if (! g_file_replace_contents (file, contents, strlen (contents), NULL, TRUE,
            G_FILE_CREATE_NONE, NULL, NULL, &err)) {
65        g_printerr ("%s.\n", err->message);
66        g_clear_error (&err);
67      }
68      g_free (contents);
69      g_object_unref (file);
70    }
71    return FALSE;
72  }
73
74  static void
75  app_activate (GApplication *app) {
76    g_print ("You␣need␣to␣give␣filenames␣as␣arguments.\n");
77  }
78
79  static void
80  app_open (GApplication *app, GFile ** files, gint n_files, gchar *hint) {
81    GtkWidget *win;
82    GtkWidget *nb;
83    GtkWidget *lab;
84    GtkNotebookPage *nbp;
85    GtkWidget *scr;
86    GtkWidget *tv;
87    GtkTextBuffer *tb;
88    char *contents;
89    gsize length;
90    char *filename;
91    int i;
92    GError *err = NULL;
93
94    win = gtk_application_window_new (GTK_APPLICATION (app));
95    gtk_window_set_title (GTK_WINDOW (win), "file␣editor");
96    gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
97
98    nb = gtk_notebook_new ();
99    gtk_window_set_child (GTK_WINDOW (win), nb);
100
101   for (i = 0; i < n_files; i++) {
102     if (g_file_load_contents (files[i], NULL, &contents, &length, NULL, &err)) {
103       scr = gtk_scrolled_window_new ();
104       tv = tfe_text_view_new ();
105       tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
106       gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
107       gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
108
109       tfe_text_view_set_file (TFE_TEXT_VIEW (tv),  g_file_dup (files[i]));
110       gtk_text_buffer_set_text (tb, contents, length);
111       g_free (contents);
```

```
112         filename = g_file_get_basename (files[i]);
113         lab = gtk_label_new (filename);
114         gtk_notebook_append_page (GTK_NOTEBOOK (nb), scr, lab);
115         nbp = gtk_notebook_get_page (GTK_NOTEBOOK (nb), scr);
116         g_object_set (nbp, "tab-expand", TRUE, NULL);
117         g_free (filename);
118       } else {
119         g_printerr ("%s.\n", err->message);
120         g_clear_error (&err);
121       }
122     }
123     if (gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb)) > 0) {
124       g_signal_connect (win, "close-request", G_CALLBACK (before_close), nb);
125       gtk_window_present (GTK_WINDOW (win));
126     } else
127       gtk_window_destroy (GTK_WINDOW (win));
128 }
129
130 int
131 main (int argc, char **argv) {
132   GtkApplication *app;
133   int stat;
134
135   app = gtk_application_new ("com.github.ToshioCP.tfe1", G_APPLICATION_HANDLES_OPEN);
136   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
137   g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
138   stat =g_application_run (G_APPLICATION (app), argc, argv);
139   g_object_unref (app);
140   return stat;
141 }
```

- 109: The GFile pointer of the TfeTextView is set to the copy of `files[i]`, which is a GFile created with the command line argument. The GFile will be destroyed by the system later. So it needs to be copied before the assignment. `g_file_dup` duplicates the GFile. Note: GFile is *not* thread safe. Duplicating GFile avoids a trouble comes from the different thread.
- 124: The "close-request" signal is connected to `before_close` handler. The fourth argument is called "user data" and it will be the second argument of the signal handler. So, `nb` is given to `before_close` as the second argument.

Now it's time to compile and run.

```
$ cd src/tfe
$ comp tfe1
$ ./a.out taketori.txt`.
```

Modify the contents and close the window. Make sure that the file is modified.

Now we got a very simple editor. It's not smart. We need more features like open, save, saveas, change font and so on. We will add them in the next section and after.

9 GtkBuilder and UI file

9.1 New, Open and Save button

We made very simple editor in the previous section. It reads files at the start and writes them out at the end of the program. It works, but is not so good. It would be better if we had "New", "Open", "Save" and "Close" buttons. This section describes how to put those buttons into the window.

The screenshot above shows the layout. The function `app_open` in the source code `tfe2.c` is as follows.

```
1 static void
2 app_open (GApplication *app, GFile ** files, gint n_files, gchar *hint) {
3   GtkWidget *win;
4   GtkWidget *nb;
```

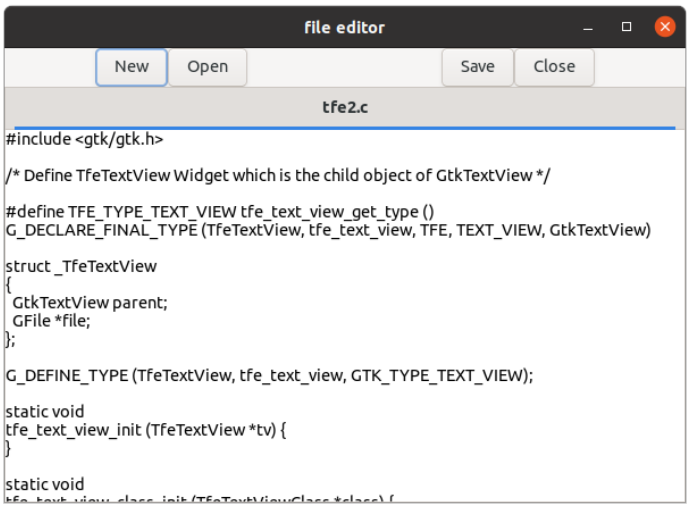


Figure 12: Screenshot of the file editor

```
5   GtkWidget *lab;
6   GtkNotebookPage *nbp;
7   GtkWidget *scr;
8   GtkWidget *tv;
9   GtkTextBuffer *tb;
10  char *contents;
11  gsize length;
12  char *filename;
13  int i;
14  GError *err = NULL;
15
16  GtkWidget *boxv;
17  GtkWidget *boxh;
18  GtkWidget *dmy1;
19  GtkWidget *dmy2;
20  GtkWidget *dmy3;
21  GtkWidget *btnn; /* button for new */
22  GtkWidget *btno; /* button for open */
23  GtkWidget *btns; /* button for save */
24  GtkWidget *btnc; /* button for close */
25
26  win = gtk_application_window_new (GTK_APPLICATION (app));
27  gtk_window_set_title (GTK_WINDOW (win), "file␣editor");
28  gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
29
30  boxv = gtk_box_new (GTK_ORIENTATION_VERTICAL, 0);
31  gtk_window_set_child (GTK_WINDOW (win), boxv);
32
33  boxh = gtk_box_new (GTK_ORIENTATION_HORIZONTAL, 0);
34  gtk_box_append (GTK_BOX (boxv), boxh);
35
36  dmy1 = gtk_label_new(NULL); /* dummy label for left space */
37  gtk_label_set_width_chars (GTK_LABEL (dmy1), 10);
38  dmy2 = gtk_label_new(NULL); /* dummy label for center space */
39  gtk_widget_set_hexpand (dmy2, TRUE);
40  dmy3 = gtk_label_new(NULL); /* dummy label for right space */
41  gtk_label_set_width_chars (GTK_LABEL (dmy3), 10);
42  btnn = gtk_button_new_with_label ("New");
43  btno = gtk_button_new_with_label ("Open");
44  btns = gtk_button_new_with_label ("Save");
45  btnc = gtk_button_new_with_label ("Close");
46
```

```
47    gtk_box_append (GTK_BOX (boxh), dmy1);
48    gtk_box_append (GTK_BOX (boxh), btnn);
49    gtk_box_append (GTK_BOX (boxh), btno);
50    gtk_box_append (GTK_BOX (boxh), dmy2);
51    gtk_box_append (GTK_BOX (boxh), btns);
52    gtk_box_append (GTK_BOX (boxh), btnc);
53    gtk_box_append (GTK_BOX (boxh), dmy3);
54
55    nb = gtk_notebook_new ();
56    gtk_widget_set_hexpand (nb, TRUE);
57    gtk_widget_set_vexpand (nb, TRUE);
58    gtk_box_append (GTK_BOX (boxv), nb);
59
60    for (i = 0; i < n_files; i++) {
61      if (g_file_load_contents (files[i], NULL, &contents, &length, NULL, &err)) {
62        scr = gtk_scrolled_window_new ();
63        tv = tfe_text_view_new ();
64        tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
65        gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
66        gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
67
68        tfe_text_view_set_file (TFE_TEXT_VIEW (tv),  g_file_dup (files[i]));
69        gtk_text_buffer_set_text (tb, contents, length);
70        g_free (contents);
71        filename = g_file_get_basename (files[i]);
72        lab = gtk_label_new (filename);
73        gtk_notebook_append_page (GTK_NOTEBOOK (nb), scr, lab);
74        nbp = gtk_notebook_get_page (GTK_NOTEBOOK (nb), scr);
75        g_object_set (nbp, "tab-expand", TRUE, NULL);
76        g_free (filename);
77      } else {
78        g_printerr ("%s.\n", err->message);
79        g_clear_error (&err);
80      }
81    }
82    if (gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb)) > 0) {
83      gtk_window_present (GTK_WINDOW (win));
84    } else
85      gtk_window_destroy (GTK_WINDOW (win));
86  }
```

The function `app_open` builds the widgets in the main application window.

- 26-28: Creates a GtkApplicationWindow instance and sets the title and default size.
- 30-31: Creates a GtkBox instance `boxv`. It is a vertical box and a child of GtkApplicationWindow. It has two children. The first child is a horizontal box. The second child is a GtkNotebook.
- 33-34: Creates a GtkBox instance `boxh` and appends it to `boxv` as the first child.
- 36-41: Creates three dummy labels. The labels `dmy1` and `dmy3` has a character width of ten. The other label `dmy2` has hexpand property which is set to be TRUE. This makes the label expands horizontally as long as possible.
- 42-45: Creates four buttons.
- 47-53: Appends these GtkLabel and GtkButton to `boxh`.
- 55-58: Creates a GtkNotebook instance and sets hexpand and vexpand properties to be TRUE. This makes it expand horizontally and vertically as big as possible. It is appended to `boxv` as the second child.

The number of widget-build lines is 33(=58-26+1). We also needed many variables (`boxv`, `boxh`, `dmy1`, ...) and most of them used only for building the widgets. Are there any good solution to reduce these works?

Gtk provides GtkBuilder. It reads user interface (UI) data and builds a window. It reduces this cumbersome work.

9.2 The UI File

Look at the UI file `tfe3.ui` that defines widget structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <object class="GtkApplicationWindow" id="win">
    <property name="title">file editor</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
    <child>
      <object class="GtkBox">
        <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
        <child>
          <object class="GtkBox">
            <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
            <child>
              <object class="GtkLabel">
                <property name="width-chars">10</property>
              </object>
            </child>
            <child>
              <object class="GtkButton">
                <property name="label">New</property>
              </object>
            </child>
            <child>
              <object class="GtkButton">
                <property name="label">Open</property>
              </object>
            </child>
            <child>
              <object class="GtkLabel">
                <property name="hexpand">TRUE</property>
              </object>
            </child>
            <child>
              <object class="GtkButton">
                <property name="label">Save</property>
              </object>
            </child>
            <child>
              <object class="GtkButton">
                <property name="label">Close</property>
              </object>
            </child>
            <child>
              <object class="GtkLabel">
                <property name="width-chars">10</property>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkNotebook" id="nb">
            <property name="hexpand">TRUE</property>
            <property name="vexpand">TRUE</property>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

The is a XML file. Tags begin with < and end with >. There are two types of tags, the start tag and the end

tag. For example, `<interface>` is a start tag and `</interface>` is an end tag. The UI file begins and ends with interface tags. Some tags, for example object tags, can have a class and id attributes in their start tag.

- 1: XML declaration. It specifies that the XML version is 1.0 and the encoding is UTF-8.
- 3-6: An object tag with `GtkApplicationWindow` class and `win` id. This is the top level window. And the three properties of the window are defined. The `title` property is "file editor", `default-width` property is 600 and `default-height` property is 400.
- 7: Child tag means a child widget. For example, line 7 tells us that GtkBox object is a child widget of `win`.

Compare this ui file and the lines 26-58 in the `app_open` function of `tfe2.c`. Both builds the same window with its descendant widgets.

You can check the ui file with `gtk4-builder-tool`.

- `gtk4-builder-tool validate <ui file name>` validates the ui file. If the ui file includes some syntactical error, `gtk4-builder-tool` prints the error.
- `gtk4-builder-tool simplify <ui file name>` simplifies the ui file and prints the result. If `--replace` option is given, it replaces the ui file with the simplified one. If the ui file specifies a value of property but it is default, then it will be removed. For example, the default orientation is horizontal so the simplification removes line 12. And some values are simplified. For example, "TRUE"and "FALSE" becomes "1" and "0" respectively. However, "TRUE" or "FALSE" is better for maintenance.

It is a good idea to check your ui file before compiling.

9.3 GtkBuilder

GtkBuilder builds widgets based on a ui file.

```
GtkBuilder *build;

build = gtk_builder_new_from_file ("tfe3.ui");
win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
nb = GTK_WIDGET (gtk_builder_get_object (build, "nb"));
g_object_unref(build);
```

The function `gtk_builder_new_from_file` reads the file `tfe3.ui`. Then, it builds the widgets and creates GtkBuilder object. All the widgets are connected based on the parent-children relationship described in the ui file. We can retrieve objects from the builder object with `gtk_builder_get_object` function. The top level window, its id is "win" in the ui file, is taken and assigned to the variable `win`, the application property of which is set to `app` with the `gtk_window_set_application` function. GtkNotebook with the id "nb" in the ui file is also taken and assigned to the variable `nb`. After the window and application are connected, GtkBuilder instance is useless. It is released with `g_object_unref` function.

The ui file reduces lines in the C source file.

```
$ cd tfe; diff tfe2.c tfe3.c
59a60
>   GtkBuilder *build;
61,104c62,66
<   GtkWidget *boxv;
<   GtkWidget *boxh;
<   GtkWidget *dmy1;
<   GtkWidget *dmy2;
<   GtkWidget *dmy3;
<   GtkWidget *btnn; /* button for new */
<   GtkWidget *btno; /* button for open */
<   GtkWidget *btns; /* button for save */
<   GtkWidget *btnc; /* button for close */
<
<   win = gtk_application_window_new (GTK_APPLICATION (app));
<   gtk_window_set_title (GTK_WINDOW (win), "file editor");
<   gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
<
```

```
<   boxv = gtk_box_new (GTK_ORIENTATION_VERTICAL, 0);
<   gtk_window_set_child (GTK_WINDOW (win), boxv);
<
<   boxh = gtk_box_new (GTK_ORIENTATION_HORIZONTAL, 0);
<   gtk_box_append (GTK_BOX (boxv), boxh);
<
<   dmy1 = gtk_label_new(NULL); /* dummy label for left space */
<   gtk_label_set_width_chars (GTK_LABEL (dmy1), 10);
<   dmy2 = gtk_label_new(NULL); /* dummy label for center space */
<   gtk_widget_set_hexpand (dmy2, TRUE);
<   dmy3 = gtk_label_new(NULL); /* dummy label for right space */
<   gtk_label_set_width_chars (GTK_LABEL (dmy3), 10);
<   btnn = gtk_button_new_with_label ("New");
<   btno = gtk_button_new_with_label ("Open");
<   btns = gtk_button_new_with_label ("Save");
<   btnc = gtk_button_new_with_label ("Close");
<
<   gtk_box_append (GTK_BOX (boxh), dmy1);
<   gtk_box_append (GTK_BOX (boxh), btnn);
<   gtk_box_append (GTK_BOX (boxh), btno);
<   gtk_box_append (GTK_BOX (boxh), dmy2);
<   gtk_box_append (GTK_BOX (boxh), btns);
<   gtk_box_append (GTK_BOX (boxh), btnc);
<   gtk_box_append (GTK_BOX (boxh), dmy3);
<
<   nb = gtk_notebook_new ();
<   gtk_widget_set_hexpand (nb, TRUE);
<   gtk_widget_set_vexpand (nb, TRUE);
<   gtk_box_append (GTK_BOX (boxv), nb);
<
---
>   build = gtk_builder_new_from_file ("tfe3.ui");
>   win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
>   gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
>   nb = GTK_WIDGET (gtk_builder_get_object (build, "nb"));
>   g_object_unref(build);
138c100
<   app = gtk_application_new ("com.github.ToshioCP.tfe2",
    G_APPLICATION_HANDLES_OPEN);
---
>   app = gtk_application_new ("com.github.ToshioCP.tfe3",
    G_APPLICATION_HANDLES_OPEN);
144a107
>
```

`61,104c62,66` means that 44 (=104-61+1) lines are changed to 5 (=66-62+1) lines. Therefore, 39 lines are reduced. Using ui file not only shortens C source files, but also makes the widgets structure clear.

Now I'll show you `app_open` function in the C file `tfe3.c`.

```
1  static void
2  app_open (GApplication *app, GFile ** files, gint n_files, gchar *hint) {
3    GtkWidget *win;
4    GtkWidget *nb;
5    GtkWidget *lab;
6    GtkNotebookPage *nbp;
7    GtkWidget *scr;
8    GtkWidget *tv;
9    GtkTextBuffer *tb;
10   char *contents;
11   gsize length;
12   char *filename;
13   int i;
14   GError *err = NULL;
15   GtkBuilder *build;
```

```
16
17    build = gtk_builder_new_from_file ("tfe3.ui");
18    win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
19    gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
20    nb = GTK_WIDGET (gtk_builder_get_object (build, "nb"));
21    g_object_unref(build);
22    for (i = 0; i < n_files; i++) {
23      if (g_file_load_contents (files[i], NULL, &contents, &length, NULL, &err)) {
24        scr = gtk_scrolled_window_new ();
25        tv = tfe_text_view_new ();
26        tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
27        gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
28        gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
29
30        tfe_text_view_set_file (TFE_TEXT_VIEW (tv),  g_file_dup (files[i]));
31        gtk_text_buffer_set_text (tb, contents, length);
32        g_free (contents);
33        filename = g_file_get_basename (files[i]);
34        lab = gtk_label_new (filename);
35        gtk_notebook_append_page (GTK_NOTEBOOK (nb), scr, lab);
36        nbp = gtk_notebook_get_page (GTK_NOTEBOOK (nb), scr);
37        g_object_set (nbp, "tab-expand", TRUE, NULL);
38        g_free (filename);
39      } else {
40        g_printerr ("%s.\n", err->message);
41        g_clear_error (&err);
42      }
43    }
44    if (gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb)) > 0) {
45      gtk_window_present (GTK_WINDOW (win));
46    } else
47      gtk_window_destroy (GTK_WINDOW (win));
48  }
```

The whole source code of `tfe3.c` is stored in the src/tfe directory.

9.3.1 Using ui string

GtkBuilder can build widgets with string. Use `gtk_builder_new_from_string` instead of `gtk_builder_new_from_file`.

```
char *uistring;

uistring =
"<interface>"
  "<object␣class="GtkApplicationWindow"␣id="win">"
    "<property␣name=\"title\">file␣editor</property>"
    "<property␣name=\"default-width\">600</property>"
    "<property␣name=\"default-height\">400</property>"
    "<child>"
      "<object␣class=\"GtkBox\">"
        "<property␣name="orientation">GTK_ORIENTATION_VERTICAL</property>"
... ... ...
... ... ...
"</interface>";

build = gtk_builder_new_from_string (uistring, -1);
```

This method has an advantage and disadvantage. The advantage is that the ui string is written in the source code. So, no ui file is needed on runtime. The disadvantage is that writing C string is a bit bothersome because of the double quotes. If you want to use this method, you should write a script that transforms ui file into C-string.

- Add backslash before each double quote.
- Add double quotes at the left and right of the string in each line.

9.3.2 Gresource

Gresource is similar to string. But Gresource is compressed binary data, not text data. And there's a compiler that compiles ui file into Gresource. It can compile not only text files but also binary files such as images, sounds and so on. And after compilation, it bundles them up into one Gresource object.

An xml file is necessary for the resource compiler `glib-compile-resources`. It describes resource files.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <gresources>
3   <gresource prefix="/com/github/ToshioCP/tfe3">
4     <file>tfe3.ui</file>
5   </gresource>
6 </gresources>
```

- 2: `gresources` tag can include multiple gresources (gresource tags). However, this xml has only one gresource.
- 3: The gresource has a prefix `/com/github/ToshioCP/tfe3`.
- 4: The name of the gresource is `tfe3.ui`. The resource will be pointed with `/com/github/ToshioCP/tfe3/tfe3.ui` by GtkBuilder. The pattern is "prefix" + "name". If you want to add more files, insert them between line 4 and 5.

Save this xml text to `tfe3.gresource.xml`. The gresource compiler `glib-compile-resources` shows its usage with the argument `--help`.

```
$ glib-compile-resources --help
Usage:
  glib-compile-resources [OPTION..] FILE

Compile a resource specification into a resource file.
Resource specification files have the extension .gresource.xml,
and the resource file have the extension called .gresource.

Help Options:
  -h, --help                    Show help options

Application Options:
  --version                     Show program version and exit
  --target=FILE                 Name of the output file
  --sourcedir=DIRECTORY         The directories to load files referenced in FILE from
      (default: current directory)
  --generate                    Generate output in the format selected for by the
      target filename extension
  --generate-header             Generate source header
  --generate-source             Generate source code used to link in the resource
      file into your code
  --generate-dependencies       Generate dependency list
  --dependency-file=FILE        Name of the dependency file to generate
  --generate-phony-targets      Include phony targets in the generated dependency file
  --manual-register             Don't automatically create and register resource
  --internal                    Don't export functions; declare them G_GNUC_INTERNAL
  --external-data               Don't embed resource data in the C file; assume it's
      linked externally instead
  --c-name                      C identifier name used for the generated source code
  -C, --compiler                The target C compiler (default: the CC environment
      variable)
```

Now run the compiler.

```
$ glib-compile-resources tfe3.gresource.xml --target=resources.c --generate-source
```

Then a C source file `resources.c` is generated. Modify `tfe3.c` and save it as `tfe3_r.c`.

```
#include "resources.c"
... ... ...
... ... ...
```

```
build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfe3/tfe3.ui");
... ... ...
... ... ...
```

The function `gtk_builder_new_from_resource` builds widgets from a resource.

Then, compile and run it.

```
$ comp tfe3_r
$ ./a.out tfe2.c
```

A window appears and it is the same as the screenshot at the beginning of this page.

Generally, resource is the best way for C language. If you use other languages like Ruby, string is better than resource.

10 Build system

10.1 Managing big source files

We've compiled a small editor so far. The program is also small and not complicated yet. But if it grows bigger, it will be difficult to maintain. So, we should do the followings now.

- We've had only one C source file and put everything in it. We need to divide it and sort them out.
- There are two compilers, `gcc` and `glib-compile-resources`. We should control them by one building tool.

10.2 Divide a C source file into two parts.

When you divide C source file into several parts, each file should contain one thing. For example, our source has two things, the definition of TfeTextView and functions related to GtkApplication and GtkApplicationWindow. It is a good idea to separate them into two files, `tfetextview.c` and `tfe.c`.

- `tfetextview.c` includes the definition and functions of TfeTextView.
- `tfe.c` includes functions like `main`, `app_activate`, `app_open` and so on, which relate to GtkApplication and GtkApplicationWindow

Now we have three source files, `tfetextview.c`, `tfe.c` and `tfe3.ui`. The `3` of `tfe3.ui` is like a version number. Managing version with filenames is one possible idea but it also has a problem. You need to rewrite filename in each version and it affects to contents of source files that refer to filenames. So, we should take `3` away from the filename.

In `tfe.c` the function `tfe_text_view_new` is invoked to create a TfeTextView instance. But it is defined in `tfetextview.c`, not `tfe.c`. The lack of the declaration (not definition) of `tfe_text_view_new` makes error when `tfe.c` is compiled. The declaration is necessary in `tfe.c`. Those public information is usually written in header files. It has `.h` suffix like `tfetextview.h`. And header files are included by C source files. For example, `tfetextview.h` is included by `tfe.c`.

The source files are shown below.

`tfetextview.h`

```
1  #include <gtk/gtk.h>
2
3  #define TFE_TYPE_TEXT_VIEW tfe_text_view_get_type ()
4  G_DECLARE_FINAL_TYPE (TfeTextView, tfe_text_view, TFE, TEXT_VIEW, GtkTextView)
5
6  void
7  tfe_text_view_set_file (TfeTextView *tv, GFile *f);
8
9  GFile *
10 tfe_text_view_get_file (TfeTextView *tv);
11
12 GtkWidget *
13 tfe_text_view_new (void);
```

tfetextview.c

```c
#include <gtk/gtk.h>
#include "tfetextview.h"

struct _TfeTextView
{
  GtkTextView parent;
  GFile *file;
};

G_DEFINE_TYPE (TfeTextView, tfe_text_view, GTK_TYPE_TEXT_VIEW);

static void
tfe_text_view_init (TfeTextView *tv) {
}

static void
tfe_text_view_class_init (TfeTextViewClass *class) {
}

void
tfe_text_view_set_file (TfeTextView *tv, GFile *f) {
  tv->file = f;
}

GFile *
tfe_text_view_get_file (TfeTextView *tv) {
  return tv->file;
}

GtkWidget *
tfe_text_view_new (void) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_TEXT_VIEW, NULL));
}
```

tfe.c

```c
#include <gtk/gtk.h>
#include "tfetextview.h"

static void
app_activate (GApplication *app) {
  g_print ("You need a filename argument.\n");
}

static void
app_open (GApplication *app, GFile ** files, gint n_files, gchar *hint) {
  GtkWidget *win;
  GtkWidget *nb;
  GtkWidget *lab;
  GtkNotebookPage *nbp;
  GtkWidget *scr;
  GtkWidget *tv;
  GtkTextBuffer *tb;
  char *contents;
  gsize length;
  char *filename;
  int i;
  GError *err = NULL;
  GtkBuilder *build;

  build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfe/tfe.ui");
  win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
  gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
```

```
28    nb = GTK_WIDGET (gtk_builder_get_object (build, "nb"));
29    g_object_unref (build);
30    for (i = 0; i < n_files; i++) {
31      if (g_file_load_contents (files[i], NULL, &contents, &length, NULL, &err)) {
32        scr = gtk_scrolled_window_new ();
33        tv = tfe_text_view_new ();
34        tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
35        gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
36        gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
37
38        tfe_text_view_set_file (TFE_TEXT_VIEW (tv),  g_file_dup (files[i]));
39        gtk_text_buffer_set_text (tb, contents, length);
40        g_free (contents);
41        filename = g_file_get_basename (files[i]);
42        lab = gtk_label_new (filename);
43        gtk_notebook_append_page (GTK_NOTEBOOK (nb), scr, lab);
44        nbp = gtk_notebook_get_page (GTK_NOTEBOOK (nb), scr);
45        g_object_set (nbp, "tab-expand", TRUE, NULL);
46        g_free (filename);
47      } else {
48        g_printerr ("%s.\n", err->message);
49        g_clear_error (&err);
50      }
51    }
52    if (gtk_notebook_get_n_pages (GTK_NOTEBOOK (nb)) > 0) {
53      gtk_window_present (GTK_WINDOW (win));
54    } else
55      gtk_window_destroy (GTK_WINDOW (win));
56  }
57
58  int
59  main (int argc, char **argv) {
60    GtkApplication *app;
61    int stat;
62
63    app = gtk_application_new ("com.github.ToshioCP.tfe", G_APPLICATION_HANDLES_OPEN);
64    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
65    g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
66    stat =g_application_run (G_APPLICATION (app), argc, argv);
67    g_object_unref (app);
68    return stat;
69  }
```

The ui file `tfe.ui` is the same as `tfe3.ui` in the previous section.

`tfe.gresource.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <gresources>
3    <gresource prefix="/com/github/ToshioCP/tfe">
4      <file>tfe.ui</file>
5    </gresource>
6  </gresources>
```

Dividing a file makes it easy to maintain. But now we face a new problem. The building step increases.

- Compiling the ui file `tfe.ui` into `resources.c`.
- Compiling `tfe.c` into `tfe.o` (object file).
- Compiling `tfetextview.c` into `tfetextview.o`.
- Compiling `resources.c` into `resources.o`.
- Linking all the object files into application `tfe`.

Build tools manage the steps.

10.3 Meson and Ninja

I'll explain Meson and Ninja build tools.

Other possible tools are Make and Autotools. They are traditional tools but slower than Ninja. So, many developers use Meson and Ninja lately. For example, GTK 4 uses them.

You need to create `meson.build` file first.

```
1  project('tfe', 'c')
2
3  gtkdep = dependency('gtk4')
4
5  gnome=import('gnome')
6  resources = gnome.compile_resources('resources','tfe.gresource.xml')
7
8  sourcefiles=files('tfe.c', 'tfetextview.c')
9
10 executable('tfe', sourcefiles, resources, dependencies: gtkdep, install: false)
```

- 1: The function `project` defines things about the project. The first argument is the name of the project and the second is the programming language.
- 3: The function `dependency` defines a dependency that is taken by `pkg-config`. We put `gtk4` as an argument.
- 5: The function `import` imports a module. In line 5, the gnome module is imported and assigned to the variable `gnome`. The gnome module provides helper tools to build GTK programs.
- 6: The method `.compile_resources` is of the gnome module and compiles files to resources under the instruction of xml file. In line 6, the resource filename is `resources`, which means `resources.c` and `resources.h`, and xml file is `tfe.gresource.xml`. This method generates C source file by default.
- 8: Defines source files.
- 10: Executable function generates a target file by compiling source files. The first argument is the filename of the target. The following arguments are source files. The last two arguments have keys and values. For example, the fourth argument has a key `dependencies` , a delimiter (`:`) and a value `gtkdep`. This type of parameter is called *keyword parameter* or *kwargs*. The value `gtkdep` is defined in line 3. The last argument tells that this project doesn't install the executable file. So it is just compiled in the build directory.

Now run meson and ninja.

```
$ meson setup _build
$ ninja -C _build
```

meson has two arguments.

- setup: The first argument is a command of meson. Setup is the default, so you can leave it out. But it is recommended to write it explicitly since version 0.64.0.
- The second argument is the name of the build directory.

Then, the executable file `tfe` is generated under the directory `_build`.

```
$ _build/tfe tfe.c tfetextview.c
```

A window appears. It includes a notebook with two pages. One is `tfe.c` and the other is `tfetextview.c`.

For further information, see The Meson Build system.

11 Instance Initialization and destruction

A new version of the text file editor (`tfe`) will be made in this section and the following four sections. It is `tfe5`. There are many changes from the prior version. They are located in two directories, src/tfe5 and src/tfetextview.

11.1 Encapsulation

We've divided C source file into two parts. But it is not enough in terms of encapsulation.

- `tfe.c` includes everything other than TfeTextView. It should be divided into at least two parts, `tfeapplication.c` and `tfenotebook.c`.
- Header files also need to be organized.

However, first of all, I'd like to focus on the object TfeTextView. It is a child object of GtkTextView and has a new member `file` in it. The important thing is to manage the Gfile object pointed by `file`.

- What is necessary to GFile when creating (or initializing) TfeTextView?
- What is necessary to GFile when destructing TfeTextView?
- Should TfeTextView read/write a file by itself or not?
- How it communicates with objects outside?

You need to know at least class, instance and signals before thinking about them. I will explain them in this section and the next section. After that I will explain:

- Organizing functions.
- How to use GtkFileDialog. It is a new class made in the version 4.10 and replaces GtkFileChooserDialog.

11.2 GObject and its children

GObject and its children are objects, which have both class and object C structures. First, think about instances. An instance is memories which has the object structure. The following is the structure of TfeTextView.

```
/* This typedef statement is automatically generated by the macro
    G_DECLARE_FINAL_TYPE */
typedef struct _TfeTextView TfeTextView;

struct _TfeTextView {
  GtkTextView parent;
  GFile *file;
};
```

The members of the structure are:

- The member `parent` is a GtkTextView C structure. It is declared in `gtktextview.h`. GtkTextView is the parent of TfeTextView.
- The member `file` is a pointer to a GFile. It can be NULL if the TfeTextView instance has no file. The most common case is that the instance is newly created.

You can find the declaration of the structures of the ancestors in the source files in GTK or GLib. The following is extracted from the source files (not exactly the same).

```
typedef struct _GObject GObject;
typedef struct _GObject GInitiallyUnowned;
struct  _GObject
{
  GTypeInstance  g_type_instance;
  volatile guint ref_count;
  GData         *qdata;
};

typedef struct _GtkWidget GtkWidget;
struct _GtkWidget
{
  GInitiallyUnowned parent_instance;
  GtkWidgetPrivate *priv;
};

typedef struct _GtkTextView GtkTextView;
struct _GtkTextView
```

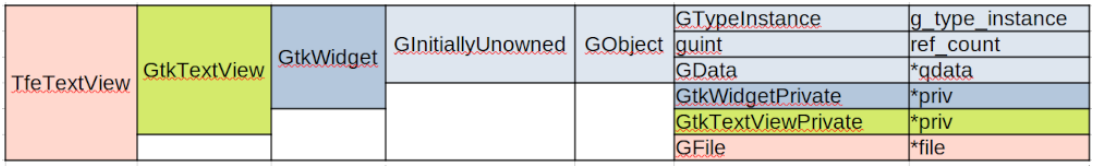


Figure 13: The structure of the instance TfeTextView

```
{
  GtkWidget parent_instance;
  GtkTextViewPrivate *priv;
};
```

In each structure, its parent is declared at the top of the members. So, all the ancestors are included in the child object. The structure of `TfeTextView` is like the following diagram.

Derivable classes (ancestor classes) have their own private data area which are not included by the structure above. For example, GtkWidget has GtkWidgetPrivate (C structure) for its private data.

Notice declarations are not definitions. So, no memories are allocated when C structures are declared. Memories are allocated to them from the heap area when the `tfe_text_view_new` function is called. At the same time, the ancestors' private area allocated for the TfeTetView. They are hidden from TfeTextView and it can't access to them directly. The created memory is called instance. When a TfeTextView instance is created, it is given three data area.

- The instance (C structure).
- GtkWidgetPrivate structure.
- GtkTextViewPrivate structure.

TfeTextView functions can access to its instance only. The GtkWidgetPrivate and GtkTextViewPrivate are used by the ancestors' functions. See the following example.

```
GtkWidget *tv = tfe_text_view_new ();
GtkTextBuffer *buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
```

The parent's function `gtk_text_view_get_buffer` accesses the GtkTextViewPrivate data (owned by `tv`). There is a pointer, which points the GtkBuffer, in the private area and the function returns the pointer. (Actual behavior is a bit more complicated.)

TfeTextView instances inherit the ancestors functions like this.

A TfeTextView instance is created every time the `tfe_text_view_new` function is called. Therefore, multiple TfeTextView instances can exist.

11.3 Initialization of TfeTextView instances

The function `tfe_text_view_new` creates a new TfeTextView instance.

```
1 GtkWidget *
2 tfe_text_view_new (void) {
3   return GTK_WIDGET (g_object_new (TFE_TYPE_TEXT_VIEW, "wrap-mode",
        GTK_WRAP_WORD_CHAR, NULL));
4 }
```

When this function is invoked, a TfeTextView instance is created and initialized. The initialization process is as follows.

1. When the instance is created, GtkWidgetPrivate and GtkTextViewPrivate structures are also created
2. Initializes GObject (GInitiallyUnowned) part in the TfeTextView instance.
3. Initializes GtkWidget part (the first `priv`) in the TfeTextView instance and GtkWidgetPrivate structure.
4. Initializes GtkTextView part (the second `priv`) in the TfeTextView instance and GtkTextViewPrivate structure.
5. Initializes TfeTextView part (`file`) in the TfeTextView instance.

The step two through four is done by `g_object_init`, `gtk_widget_init` and `gtk_text_view_init`. They are called by the system automatically and you don't need to care about them. Step five is done by the function `tfe_text_view_init` in `tfetextview.c`.

```
1 static void
2 tfe_text_view_init (TfeTextView *tv) {
3   tv->file = NULL;
4 }
```

This function just initializes `tv->file` to be `NULL`.

11.4 Functions and Classes

In Gtk, all objects derived from GObject have classes and instances (but abstract objects have only classes). Instances are memory of C structure, which are described in the previous two subsections. Each object can have more than one instance. Those instances have the same structure. Instances just have data. Therefore, it doesn't define object's behavior. We need at least two things. One is functions and the other is class methods.

The latest GTK 4 document classifies functions into a constructor, functions and instance methods.

- constructors: Their name are always `gtk_(objectname)_new`. They create the objects.
- functions: Their first parameter (argument) is *NOT* the instance. Usually functions are utility functions for the class.
- instance methods: Their first parameter (argument) is the instance. They do some task for the specific instance.

This tutorial uses `functions` in two ways, broad or narrow sense.

You've already seen many functions. For example,

- `TfeTextView *tfe_text_view_new (void);` is a function (constructor) to create a TfeTextView instance.
- `GtkTextBuffer *gtk_text_view_get_buffer (GtkTextView *textview)` is a function (instance method) to get a GtkTextBuffer from GtkTextView.

Functions are public, which means that they are expected to be used by other objects. They are similar to public methods in object oriented languages.

Class (C structure) mainly consists of pointers to C functions. They are called *class methods* and used by the object itself or its descendant objects. For example, GObject class is declared in `gobject.h` in GLib source files.

```
1  typedef struct _GObjectClass             GObjectClass;
2  typedef struct _GObjectClass             GInitiallyUnownedClass;
3
4  struct  _GObjectClass
5  {
6    GTypeClass   g_type_class;
7
8    /*< private >*/
9    GSList      *construct_properties;
10
11   /*< public >*/
12   /* seldom overridden */
13   GObject*   (*constructor)     (GType                  type,
14                                  guint                  n_construct_properties,
15                                  GObjectConstructParam *construct_properties);
16   /* overridable methods */
17   void       (*set_property)        (GObject        *object,
18                                      guint           property_id,
19                                      const GValue   *value,
20                                      GParamSpec     *pspec);
21   void       (*get_property)        (GObject        *object,
22                                      guint           property_id,
23                                      GValue         *value,
24                                      GParamSpec     *pspec);
```

```
25    void       (*dispose)          (GObject        *object);
26    void       (*finalize)         (GObject        *object);
27    /* seldom overridden */
28    void       (*dispatch_properties_changed) (GObject      *object,
29                           guint     n_pspecs,
30                           GParamSpec  **pspecs);
31    /* signals */
32    void       (*notify)           (GObject    *object,
33                           GParamSpec *pspec);
34
35    /* called when done constructing */
36    void       (*constructed)      (GObject    *object);
37
38    /*< private >*/
39    gsize     flags;
40
41    gsize         n_construct_properties;
42
43    gpointer pspecs;
44    gsize n_pspecs;
45
46    /* padding */
47    gpointer  pdummy[3];
48  };
```

There's a pointer to the function `dispose` in line 25.

```
void (*dispose) (GObject *object);
```

The declaration is a bit complicated. The asterisk before the identifier `dispose` means pointer. So, the pointer `dispose` points to a function which has one parameter, which points a GObject structure, and returns no value. In the same way, line 26 says `finalize` is a pointer to the function which has one parameter, which points a GObject structure, and returns no value.

```
void (*finalize) (GObject *object);
```

Look at the declaration of `_GObjectClass` so that you would find that most of the members are pointers to functions.

- 13: A function pointed by `constructor` is called when the instance is created. It completes the initialization of the instance.
- 25: A function pointed by `dispose` is called when the instance destructs itself. Destruction process is divided into two phases. The first one is called disposing. In this phase, the instance releases all the references to other instances. The second phase is finalizing.
- 26: A function pointed by `finalize` finishes the destruction process.
- The other pointers point to functions which are called while the instance lives.

These functions are called class methods. The methods are open to its descendants. But not open to the objects which are not the descendants.

11.5 TfeTextView class

TfeTextView class is a structure and it includes all its ancestors' classes in it. Therefore, classes have similar hierarchy to instances.

```
GObjectClass (GInitiallyUnownedClass) -- GtkWidgetClass -- GtkTextViewClass --
    TfeTextViewClass
```

The following is extracted from the source codes (not exactly the same).

```
1  struct _GtkWidgetClass
2  {
3    GInitiallyUnownedClass parent_class;
4
5    /*< public >*/
```

```
6
7    /* basics */
8    void (* show)                (GtkWidget        *widget);
9    void (* hide)                (GtkWidget        *widget);
10   void (* map)                 (GtkWidget        *widget);
11   void (* unmap)               (GtkWidget        *widget);
12   void (* realize)             (GtkWidget        *widget);
13   void (* unrealize)           (GtkWidget        *widget);
14   void (* root)                (GtkWidget        *widget);
15   void (* unroot)              (GtkWidget        *widget);
16   void (* size_allocate)       (GtkWidget           *widget,
17                                 int                  width,
18                                 int                  height,
19                                 int                  baseline);
20   void (* state_flags_changed) (GtkWidget        *widget,
21                                 GtkStateFlags     previous_state_flags);
22   void (* direction_changed)   (GtkWidget        *widget,
23                                 GtkTextDirection  previous_direction);
24
25   /* size requests */
26   GtkSizeRequestMode (* get_request_mode)              (GtkWidget      *widget);
27   void               (* measure) (GtkWidget      *widget,
28                                   GtkOrientation  orientation,
29                                   int             for_size,
30                                   int            *minimum,
31                                   int            *natural,
32                                   int            *minimum_baseline,
33                                   int            *natural_baseline);
34
35   /* Mnemonics */
36   gboolean (* mnemonic_activate)        (GtkWidget           *widget,
37                                          gboolean             group_cycling);
38
39   /* explicit focus */
40   gboolean (* grab_focus)               (GtkWidget           *widget);
41   gboolean (* focus)                    (GtkWidget           *widget,
42                                          GtkDirectionType     direction);
43   void     (* set_focus_child)          (GtkWidget           *widget,
44                                          GtkWidget           *child);
45
46   /* keyboard navigation */
47   void     (* move_focus)               (GtkWidget           *widget,
48                                          GtkDirectionType     direction);
49   gboolean (* keynav_failed)            (GtkWidget           *widget,
50                                          GtkDirectionType     direction);
51
52   gboolean     (* query_tooltip)      (GtkWidget  *widget,
53                                        int         x,
54                                        int         y,
55                                        gboolean    keyboard_tooltip,
56                                        GtkTooltip *tooltip);
57
58   void         (* compute_expand)     (GtkWidget  *widget,
59                                        gboolean   *hexpand_p,
60                                        gboolean   *vexpand_p);
61
62   void         (* css_changed)                 (GtkWidget           *widget,
63                                                 GtkCssStyleChange    *change);
64
65   void         (* system_setting_changed)      (GtkWidget           *widget,
66                                                 GtkSystemSetting     settings);
67
68   void         (* snapshot)                    (GtkWidget           *widget,
69                                                 GtkSnapshot          *snapshot);
```

```
70
71    gboolean     (* contains)                    (GtkWidget *widget,
72                                                 double     x,
73                                                 double     y);
74
75    /*< private >*/
76
77    GtkWidgetClassPrivate *priv;
78
79    gpointer padding[8];
80  };
81
82  struct _GtkTextViewClass
83  {
84    GtkWidgetClass parent_class;
85
86    /*< public >*/
87
88    void (* move_cursor)           (GtkTextView      *text_view,
89                                    GtkMovementStep   step,
90                                    int               count,
91                                    gboolean          extend_selection);
92    void (* set_anchor)            (GtkTextView      *text_view);
93    void (* insert_at_cursor)      (GtkTextView      *text_view,
94                                    const char       *str);
95    void (* delete_from_cursor)    (GtkTextView      *text_view,
96                                    GtkDeleteType     type,
97                                    int               count);
98    void (* backspace)             (GtkTextView      *text_view);
99    void (* cut_clipboard)         (GtkTextView      *text_view);
100   void (* copy_clipboard)        (GtkTextView      *text_view);
101   void (* paste_clipboard)       (GtkTextView      *text_view);
102   void (* toggle_overwrite)      (GtkTextView      *text_view);
103   GtkTextBuffer * (* create_buffer) (GtkTextView   *text_view);
104   void (* snapshot_layer)        (GtkTextView      *text_view,
105                       GtkTextViewLayer  layer,
106                       GtkSnapshot      *snapshot);
107   gboolean (* extend_selection)  (GtkTextView            *text_view,
108                                   GtkTextExtendSelection  granularity,
109                                   const GtkTextIter      *location,
110                                   GtkTextIter            *start,
111                                   GtkTextIter            *end);
112   void (* insert_emoji)          (GtkTextView      *text_view);
113
114   /*< private >*/
115
116   gpointer padding[8];
117 };
118
119 /* The following definition is generated by the macro G_DECLARE_FINAL_TYPE */
120 typedef struct {
121   GtkTextView parent_class;
122 } TfeTextViewClass;
```

- 120-122: This three lines are generated by the macro `G_DECLARE_FINAL_TYPE`. So, they are not written in either `tfe_text_view.h` or `tfe_text_view.c`.
- 3, 84, 121: Each class has its parent class at the first member of its structure. It is the same as instance structures.
- Class members in ancestors are open to the descendant class. So, they can be changed in `tfe_text_view_class_init` function. For example, the `dispose` pointer in GObjectClass will be overridden later in `tfe_text_view_class_init`. (Override is an object oriented programming terminology. Override is rewriting ancestors' class methods in the descendant class.)
- Some class methods are often overridden. `set_property`, `get_property`, `dispose`, `finalize` and

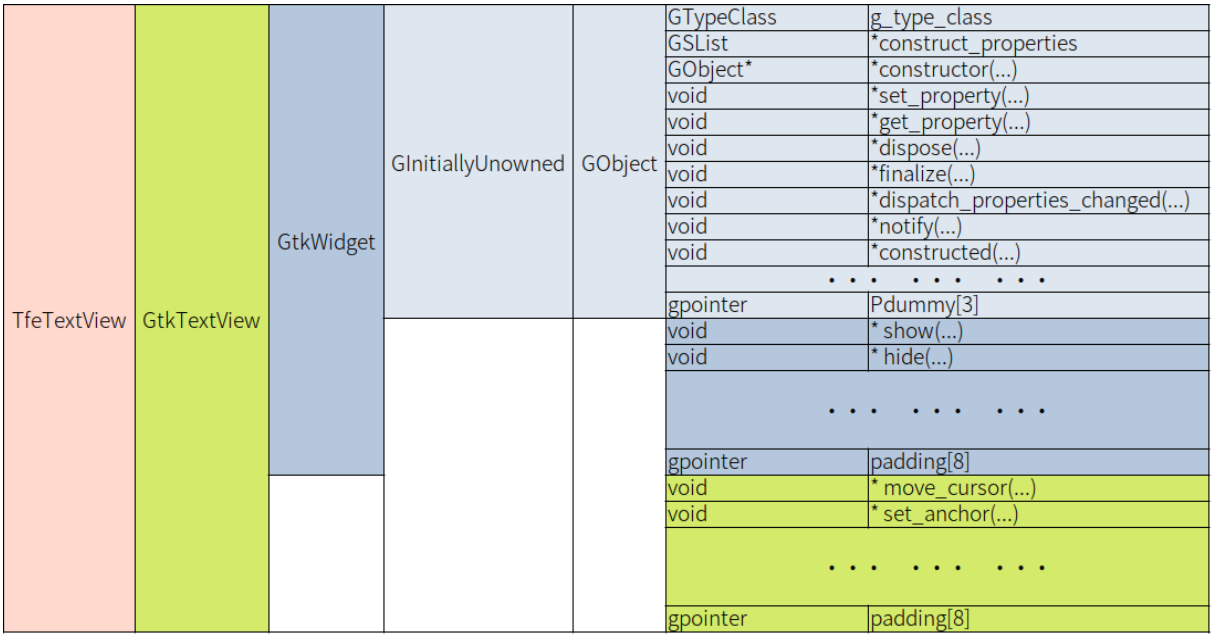


Figure 14: The structure of TfeTextView Class

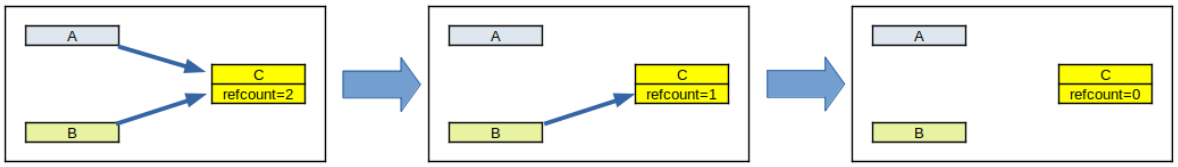


Figure 15: Reference count of B

`constructed` are such methods.

TfeTextViewClass includes its ancestors' class in it. It is illustrated in the following diagram.

11.6 Destruction of TfeTextView

Every Object derived from GObject has a reference count. If an object A refers to an object B, then A keeps a pointer to B in A and at the same time increases the reference count of B by one with the function `g_object_ref (B)`. If A doesn't need B any longer, A discards the pointer to B (usually it is done by assigning NULL to the pointer) and decreases the reference count of B by one with the function `g_object_unref (B)`.

If two objects A and B refer to C, then the reference count of C is two. If A no longer needs C, A discards the pointer to C and decreases the reference count in C by one. Now the reference count of C is one. In the same way, if B no longer needs C, B discards the pointer to C and decreases the reference count in C by one. At this moment, no object refers to C and the reference count of C is zero. This means C is no longer useful. Then C destructs itself and finally the memories allocated to C is freed.

The idea above is based on an assumption that an object referred by nothing has reference count of zero. When the reference count drops to zero, the object starts its destruction process. The destruction process is split into two phases: disposing and finalizing. In the disposing process, the object invokes the function pointed by `dispose` in its class to release all references to other instances. After that, it invokes the function pointed by `finalize` in its class to complete the destruction process.

In the destruction process, TfeTextView needs to unref the GFile pointed by `tv->file`. You must write the dispose handler `tfe_text_view_dispose`.

```
1 static void
2 tfe_text_view_dispose (GObject *gobject) {
```

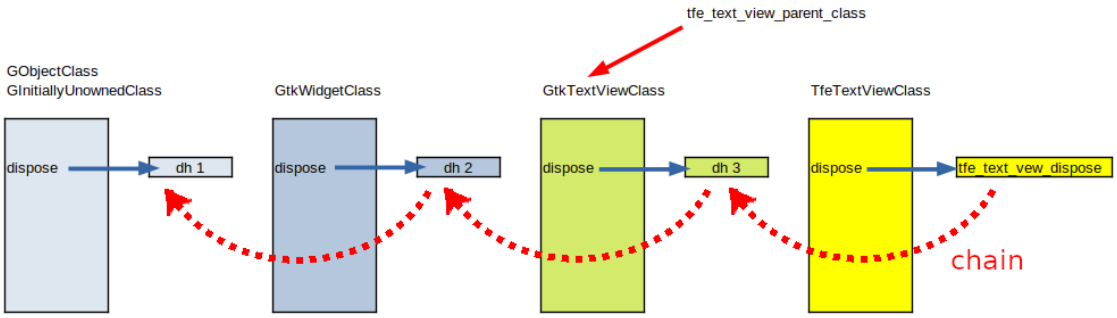


Figure 16: dispose handlers

```
3    TfeTextView *tv = TFE_TEXT_VIEW (gobject);
4
5    if (G_IS_FILE (tv->file))
6      g_clear_object (&tv->file);
7
8    G_OBJECT_CLASS (tfe_text_view_parent_class)->dispose (gobject);
9  }
```

- 5,6: If `tv->file` points a GFile, it decreases the reference count of the GFile instance. The function `g_clear_object` decreases the reference count and assigns NULL to `tv->file`. In dispose handlers, we usually use `g_clear_object` rather than `g_object_unref`.
- 8: invokes parent's dispose handler. (This will be explained later.)

In the disposing process, the object uses the pointer in its class to call the handler. Therefore, `tfe_text_view_dispose` needs to be registered in the class when the TfeTextView class is initialized. The function `tfe_text_view_class_init` is the class initialization function and it is declared in the `G_DEFINE_TYPE` macro expansion.

```
static void
tfe_text_view_class_init (TfeTextViewClass *class) {
  GObjectClass *object_class = G_OBJECT_CLASS (class);

  object_class->dispose = tfe_text_view_dispose;
}
```

Each ancestors' class has been created before TfeTextViewClass is created. Therefore, there are four classes and each class has a pointer to each dispose handler. Look at the following diagram. There are four classes – GObjectClass (GInitiallyUnownedClass), GtkWidgetClass, GtkTextViewClass and TfeTextViewClass. Each class has its own dispose handler – `dh1`, `dh2`, `dh3` and `tfe_text_view_dispose`.

Now, look at the `tfe_text_view_dispose` program above. It first releases the reference to GFile object pointed by `tv->file`. Then it invokes its parent's dispose handler in line 8.

```
G_OBJECT_CLASS (tfe_text_view_parent_class)->dispose (gobject);
```

A variable `tfe_text_view_parent_class`, which is made by `G_DEFINE_FINAL_TYPE` macro, is a pointer that points the parent object class. The variable `gobject` is a pointer to TfeTextView instance which is casted as a GObject instance. Therefore, `G_OBJECT_CLASS (tfe_text_view_parent_class)->dispose` points the handler `dh3` in the diagram above. The statement `G_OBJECT_CLASS (tfe_text_view_parent_class)->dispose (gobject)` is the same as `dh3 (gobject)`, which means it releases all the reference to the other instances in the GtkTextViewPrivate in the TfeTextView instance. After that, `dh3` calls `dh2`, and `dh2` calls `dh1`. Finally all the references are released.

12 Signals

12.1 Signals

Each object is encapsulated in Gtk programming. And it is not recommended to use global variables because they are prone to make the program complicated. So, we need something to communicate between objects. There are two ways to do so.

- Instance methods: Instance methods are functions on instances. For example, `tb = gtk_text_view_get_buffer (tv)` is an instance method on the instance `tv`. The caller requests `tv` to give `tb`, which is a GtkTextBuffer instance connected to `tv`.
- Signals: For example, `activate` signal on GApplication object. When the application is activated, the signal is emitted. Then the handler, which has been connected to the signal, is invoked.

The caller of methods or signals are usually out of the object. One of the difference between these two is that the object is active or passive. In methods, objects passively respond to the caller. In signals, objects actively send signals to handlers.

GObject signals are registered, connected and emitted.

1. Signals are registered in the class. The registration is done usually when the class is initialized. Signals can have a default handler, which is sometimes called "object method handler". It is not a user handler connected by `g_signal_connect` family functions. A default handler is always called on any instance of the class.
2. Signals are connected to handlers by the macro `g_signal_connect` or its family functions. The connection is usually done out of the object. One important thing is that signals are connected on a certain instance. Suppose there exist two GtkButton instances A, B and a function C. Even if you connected the "clicked" signal on A to C, B and C are *not* connected.
3. When Signals are emitted, the connected handlers are invoked. Signals are emitted on the instance of the class.

12.2 Signal registration

In TfeTextView, two signals are registered.

- "change-file" signal. This signal is emitted when `tv->file` is changed.
- "open-response" signal. The function `tfe_text_view_open` doesn't return the status because it can't get the status from the file chooser dialog. (Instead, the call back function gets the status.) This signal is emitted instead of the return value of the function.

A static variable or array is used to store signal ID.

```
enum {
  CHANGE_FILE,
  OPEN_RESPONSE,
  NUMBER_OF_SIGNALS
};

static guint tfe_text_view_signals[NUMBER_OF_SIGNALS];
```

Signals are registered in the class initialization function.

```
1  static void
2  tfe_text_view_class_init (TfeTextViewClass *class) {
3    GObjectClass *object_class = G_OBJECT_CLASS (class);
4
5    object_class->dispose = tfe_text_view_dispose;
6    tfe_text_view_signals[CHANGE_FILE] = g_signal_new ("change-file",
7                                    G_TYPE_FROM_CLASS (class),
8                                    G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE |
                                        G_SIGNAL_NO_HOOKS,
9                                    0 /* class offset */,
10                                   NULL /* accumulator */,
11                                   NULL /* accumulator data */,
12                                   NULL /* C marshaller */,
```

```
13                                  G_TYPE_NONE /* return_type */,
14                                  0     /* n_params */
15                                  );
16    tfe_text_view_signals[OPEN_RESPONSE] = g_signal_new ("open-response",
17                                  G_TYPE_FROM_CLASS (class),
18                                  G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE |
                                        G_SIGNAL_NO_HOOKS,
19                                  0 /* class offset */,
20                                  NULL /* accumulator */,
21                                  NULL /* accumulator data */,
22                                  NULL /* C marshaller */,
23                                  G_TYPE_NONE /* return_type */,
24                                  1     /* n_params */,
25                                  G_TYPE_INT
26                                  );
27  }
```

- 6-15: Registers “change-file” signal. `g_signal_new` function is used. The signal “change-file” has no default handler (object method handler) so the offset (the line 9) is set to zero. You usually don’t need a default handler. If you need it, use `g_signal_new_class_handler` function instead of `g_signal_new`. See GObject API Reference for further information.
- The return value of `g_signal_new` is the signal id. The type of signal id is guint, which is the same as unsigned int. It is used in the function `g_signal_emit`.
- 16-26: Registers “open-response” signal. This signal has a parameter.
- 24: Number of the parameters. “open-response” signal has one parameter.
- 25: The type of the parameter. `G_TYPE_INT` is a type of integer. Such fundamental types are described in GObject reference manual.

The handlers are declared as follows.

```
/* "change-file" signal handler */
void
user_function (TfeTextView *tv,
               gpointer user_data)

/* "open-response" signal handler */
void
user_function (TfeTextView *tv,
               TfeTextViewOpenResponseType response-id,
               gpointer user_data)
```

- The signal “change-file” doesn’t have parameter, so the handler’s arguments are a TfeTextView instance and a user data.
- The signal “open-response” signal has one parameter and its arguments are a TfeTextView instance, the signal’s parameter and user data.
- The variable `tv` points the instance on which the signal is emitted.
- The last argument `user_data` comes from the fourth argument of `g_signal_connect`.
- The `parameter` (`response-id`) comes from the fourth argument of `g_signal_emit`.

The values of the type `TfeTextViewOpenResponseType` are defined in `tfetextview.h`.

```
/* "open-response" signal response */
enum TfeTextViewOpenResponseType
{
  TFE_OPEN_RESPONSE_SUCCESS,
  TFE_OPEN_RESPONSE_CANCEL,
  TFE_OPEN_RESPONSE_ERROR
};
```

- The parameter is set to `TFE_OPEN_RESPONSE_SUCCESS` when `tfe_text_view_open` has successfully opened a file and read it.
- The parameter is set to `TFE_OPEN_RESPONSE_CANCEL` when the user has canceled.
- The parameter is set to `TFE_OPEN_RESPONSE_ERROR` when an error has occurred.

12.3 Signal connection

A signal and a handler are connected by the function macro `g_signal_connect`. There are some similar function macros like `g_signal_connect_after`, `g_signal_connect_swapped` and so on. However, `g_signal_connect` is used most often. The signals "change-file" and "open-response" are connected to their callback functions out of the TfeTextView object. Those callback functions are defined by users.

For example, callback functions are defined in `src/tfe6/tfewindow.c` and their names are `file_changed_cb` and `open_response_cb`. They will be explained later.

```
g_signal_connect (GTK_TEXT_VIEW (tv), "change-file", G_CALLBACK (file_changed_cb),
    nb);

g_signal_connect (TFE_TEXT_VIEW (tv), "open-response", G_CALLBACK
    (open_response_cb), nb);
```

12.4 Signal emission

A signal is emitted on the instance. A function `g_signal_emit` is used to emit the signal. The following lines are extracted from `src/tfetextview/tfetextview.c`. Each line comes from a different line.

```
g_signal_emit (tv, tfe_text_view_signals[CHANGE_FILE], 0);
g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0,
    TFE_OPEN_RESPONSE_SUCCESS);
g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0,
    TFE_OPEN_RESPONSE_CANCEL);
g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0, TFE_OPEN_RESPONSE_ERROR);
```

- The first argument is the instance on which the signal is emitted.
- The second argument is the signal id.
- The third argument is the detail of the signal. The signals "change-file" and "open-response" don't have details and the arguments are zero.
- The signal "change-file" doesn't have parameters, so there's no fourth argument.
- The signal "open-response" has one parameter. The fourth argument is the parameter.

13 TfeTextView class

The TfeTextView class will be finally completed in this section. The remaining topic is functions. TfeTextView functions, which are constructors and instance methods, are described in this section.

The source files are in the directory `src/tfetextview`. You can get them by downloading the repository.

13.1 tfetextview.h

The header file `tfetextview.h` provides:

- The type of TfeTextView, which is `TFE_TYPE_TEXT_VIEW`.
- The macro `G_DECLARE_FINAL_TYPE`, the expansion of which includes some useful functions and definitions.
- Constants for the `open-response` signal.
- Public functions of `tfetextview.c`. They are constructors and instance methods.

Therefore, Any programs use TfeTextView needs to include `tfetextview.h`.

```
1  #pragma once
2
3  #include <gtk/gtk.h>
4
5  #define TFE_TYPE_TEXT_VIEW tfe_text_view_get_type ()
6  G_DECLARE_FINAL_TYPE (TfeTextView, tfe_text_view, TFE, TEXT_VIEW, GtkTextView)
7
8  /* "open-response" signal response */
9  enum TfeTextViewOpenResponseType
10 {
```

```
11    TFE_OPEN_RESPONSE_SUCCESS,
12    TFE_OPEN_RESPONSE_CANCEL,
13    TFE_OPEN_RESPONSE_ERROR
14  };
15
16  GFile *
17  tfe_text_view_get_file (TfeTextView *tv);
18
19  void
20  tfe_text_view_open (TfeTextView *tv, GtkWindow *win);
21
22  void
23  tfe_text_view_save (TfeTextView *tv);
24
25  void
26  tfe_text_view_saveas (TfeTextView *tv);
27
28  GtkWidget *
29  tfe_text_view_new_with_file (GFile *file);
30
31  GtkWidget *
32  tfe_text_view_new (void);
```

- 1: The preprocessor directive `#pragma once` makes the header file be included only once. It is non-standard but widely used.
- 3: Includes gtk4 header files. The header file `gtk4` also has the same mechanism to avoid being included multiple times.
- 5-6: These two lines define TfeTextView type, its class structure and some useful definitions.
  - `TfeTextView` and `TfeTextViewClass` are declared as typedef of C structures.
  - You need to define a structure `_TfeTextView` later.
  - The class structure `_TfeTextViewClass` is defined here. You don't need to define it by yourself.
  - Convenience functions `TFE_TEXT_VIEW ()` for casting and `TFE_IS_TEXT_VIEW` for type check are defined.
- 8-14: A definition of the values of the "open-response" signal parameters.
- 16-32: Declarations of public functions on TfeTextView.

13.2 Constructors

A TfeTextView instance is created with `tfe_text_view_new` or `tfe_text_view_new_with_file`. These functions are called constructors.

```
GtkWidget *tfe_text_view_new (void);
```

It just creates a new TfeTextView instance and returns the pointer to the new instance.

```
GtkWidget *tfe_text_view_new_with_file (GFile *file);
```

It is given a Gfile object as an argument and it loads the file into the GtkTextBuffer instance, then returns the pointer to the new instance. The argument `file` is owned by the caller and the function doesn't change it. If an error occurs during the creation process, NULL will be returned.

Each function is defined as follows.

```
1   GtkWidget *
2   tfe_text_view_new_with_file (GFile *file) {
3     g_return_val_if_fail (G_IS_FILE (file), NULL);
4
5     GtkWidget *tv;
6     GtkTextBuffer *tb;
7     char *contents;
8     gsize length;
9
10    if (! g_file_load_contents (file, NULL, &contents, &length, NULL, NULL)) /* read
          error */
```

```
11      return NULL;
12
13    tv = tfe_text_view_new();
14    tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
15    gtk_text_buffer_set_text (tb, contents, length);
16    TFE_TEXT_VIEW (tv)->file = g_file_dup (file);
17    gtk_text_buffer_set_modified (tb, FALSE);
18    g_free (contents);
19    return tv;
20  }
21
22  GtkWidget *
23  tfe_text_view_new (void) {
24    return GTK_WIDGET (g_object_new (TFE_TYPE_TEXT_VIEW, "wrap-mode",
          GTK_WRAP_WORD_CHAR, NULL));
25  }
```

- 22-25: `tfe_text_view_new` function. Just returns the value from the function `g_object_new` but casts it to the pointer to GtkWidget. The function `g_object_new` creates any instances of its descendant class. The arguments are the type of the class, property list and NULL, which is the end mark of the property list. TfeTextView "wrap-mode" property has GTK_WRAP_WORD_CHAR as the default value.
- 1-20: `tfe_text_view_new_with_file` function.
- 3: `g_return_val_if_fail` is described in GLib API Reference – g_return_val_if_fail. And also GLib API Reference – Message Logging. It tests whether the argument `file` is a pointer to GFile. If it's true, the program goes on to the next line. If it's false, it returns NULL (the second argument) immediately. And at the same time it logs out the error message (usually the log is outputted to stderr or stdout). This function is used to check the programmer's error. If an error occurs, the solution is usually to change the (caller) program and fix the bug. You need to distinguish programmer's errors and runtime errors. You shouldn't use this function to find runtime errors.
- 10-11: Reads the file. If an error occurs, NULL is returned.
- 13: Calls the function `tfe_text_view_new`. The function creates TfeTextView instance and returns the pointer to the instance.
- 14: Gets the pointer to the GtkTextBuffer instance corresponds to `tv`. The pointer is assigned to `tb`
- 15: Assigns the contents read from the file to `tb`.
- 16: Duplicates `file` and sets `tv->file` to point it. GFile is *not* thread safe. The duplication makes sure that the GFile instance of `tv` keeps the file information even if the original one is changed by other thread.
- 17: The function `gtk_text_buffer_set_modified (tb, FALSE)` sets the modification flag of `tb` to FALSE. The modification flag indicates that the contents has been modified. It is used when the contents are saved. If the modification flag is FALSE, it doesn't need to save the contents.
- 18: Frees the memories pointed by `contents`.
- 19: Returns `tv`, which is a pointer to the newly created TfeTextView instance. If an error happens, NULL is returned.

13.3 Save and saveas functions

Save and saveas functions write the contents in the GtkTextBuffer to a file.

```
void tfe_text_view_save (TfeTextView *tv)
```

The function `tfe_text_view_save` writes the contents in the GtkTextBuffer to a file specified by `tv->file`. If `tv->file` is NULL, then it shows file chooser dialog and prompts the user to choose a file to save. Then it saves the contents to the file and sets `tv->file` to point the GFile instance for the file.

```
void tfe_text_view_saveas (TfeTextView *tv)
```

The function `saveas` shows a file chooser dialog and prompts the user to select a existed file or specify a new file to save. Then, the function changes `tv->file` and save the contents to the specified file. If an error occurs, it is shown to the user through the alert dialog. The error is managed only in the TfeTextView and no information is notified to the caller.

13.3.1 save_file function

```
1  static gboolean
2  save_file (GFile *file, GtkTextBuffer *tb, GtkWindow *win) {
3    GtkTextIter start_iter;
4    GtkTextIter end_iter;
5    char *contents;
6    gboolean stat;
7    GtkAlertDialog *alert_dialog;
8    GError *err = NULL;
9
10   gtk_text_buffer_get_bounds (tb, &start_iter, &end_iter);
11   contents = gtk_text_buffer_get_text (tb, &start_iter, &end_iter, FALSE);
12   stat = g_file_replace_contents (file, contents, strlen (contents), NULL, TRUE,
         G_FILE_CREATE_NONE, NULL, NULL, &err);
13   if (stat)
14     gtk_text_buffer_set_modified (tb, FALSE);
15   else {
16     alert_dialog = gtk_alert_dialog_new ("%s", err->message);
17     gtk_alert_dialog_show (alert_dialog, win);
18     g_object_unref (alert_dialog);
19     g_error_free (err);
20   }
21   g_free (contents);
22   return stat;
23 }
```

- The function `save_file` is called from `saveas_dialog_response` and `tfe_text_view_save`. This function saves the contents of the buffer to the file given as an argument. If error happens, it displays an error message. So, a caller of this function don't need to take care of errors. The class of this function is `static`. Therefore, only functions in this file (`tfetextview.c`) call this function. Such static functions usually don't have `g_return_val_if_fail` functions.
- 10-11: Gets the text contents from the buffer.
- 12: The function `g_file_replace_contents` writes the contents to the file and returns the status (true = success/ false = fail). It has many parameters, but some of them are almost always given the same values.
  - GFile* file: GFile to which the contents are saved.
  - const char* contents: contents to be saved. The string is owned by the caller.
  - gsize length: the length of the contents
  - const char* etag: entity tag. It is usually NULL.
  - gboolean make_backup: true to make a backup if the file exists. false not to make it. the file will be overwritten.
  - GFileCreateFlags flags: usually `G_FILE_CREATE_NONE` is fine.
  - char** new_etag: new entity tag. It is usually NULL.
  - GCancellable* cancellable: If a cancellable instance is set, the other thread can cancel this operation. it is usually NULL.
  - GError** error: If error happens, GError will be set.
- 13,14: If no error happens, set the modified flag to be FALSE. This means that the buffer is not modified since it has been saved.
- 16-19: If it fails to save the contents, an error message will be displayed.
- 16: Creates an alert dialog. The parameters are printf-like format string followed by values to insert into the string. GtkAlertDialog is available since version 4.10. If your version is older than 4.10, use GtkMessageDialog instead. GtkMessageDialog is deprecated since version 4.10.
- 17: Show the alert dialog. The parameters are the dialog and the transient parent window. This allows window managers to keep the dialog on top of the parent window, or center the dialog over the parent window. It is possible to give no parent window to the dialog by giving NULL as the argument. However, it is encouraged to give parents to dialogs.
- 18: Releases the dialog.
- 19: Frees the GError struct pointed by `err` with `g_error_free` function.
- 21: Frees `contents`.
- 22: Returns the status to the caller.

13.3.2 save_dialog_cb function

```
1  static void
2  save_dialog_cb(GObject *source_object, GAsyncResult *res, gpointer data) {
3    GtkFileDialog *dialog = GTK_FILE_DIALOG (source_object);
4    TfeTextView *tv = TFE_TEXT_VIEW (data);
5    GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
6    GFile *file;
7    GtkWidget *win = gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_WINDOW);
8    GError *err = NULL;
9    GtkAlertDialog *alert_dialog;
10
11   if (((file = gtk_file_dialog_save_finish (dialog, res, &err)) != NULL) &&
         save_file(file, tb, GTK_WINDOW (win))) {
12     // The following is complicated. The comments here will help your understanding
13     // G_IS_FILE(tv->file) && tv->file == file  => nothing to do
14     // G_IS_FILE(tv->file) && tv->file != file  => unref(tv->file), tv->file=file,
           emit change_file signal
15     // tv->file==NULL                           =>                  tv->file=file,
           emit change_file signal
16     if (! (G_IS_FILE (tv->file) && g_file_equal (tv->file, file))) {
17       if (G_IS_FILE (tv->file))
18         g_object_unref (tv->file);
19       tv->file = file; // The ownership of 'file' moves to TfeTextView.
20       g_signal_emit (tv, tfe_text_view_signals[CHANGE_FILE], 0);
21     }
22   }
23   if (err) {
24     alert_dialog = gtk_alert_dialog_new ("%s", err->message);
25     gtk_alert_dialog_show (alert_dialog, GTK_WINDOW (win));
26     g_object_unref (alert_dialog);
27     g_clear_error (&err);
28   }
29 }
```

- The function `save_dialog_cb` is a call back function that is given to the `gtk_file_dialog_save` function as an argument. The `gtk_file_dialog_save` shows a file chooser dialog to the user. The user chooses or types a filename and clicks on the `Save` button or just clicks on the `Cancel` button. Then the call back function is called with the result. This is the general way in GIO to manage asynchronous operations. A pair of functions `g_data_input_stream_read_line_async` and `g_data_input_stream_read_line_finish` are one example. These functions are thread-safe. The arguments of `save_dialog_cb` are:
  - GObject *source_object: The GObject instance that the operation was started with. It is actually the GtkFileDialog instance that is shown to the user. However, the call back function is defined as `AsyncReadyCallback`, which is a general call back function for an asynchronous operation. So the type is GObject and you need to cast it to GtkFileDialog later.
  - GAsyncResult *res: The result of the asynchronous operation. It will be given to the `gtk_dialog_save_finish` function.
  - gpointer data: A user data set in the `gtk_dialog_save` function.
- 11: Calls `gtk_dialog_save_finish`. It is given the result `res` as an argument and returns a pointer to a GFile object the user has chosen. If the user has canceled or an error happens, it returns NULL, creates a GError object and sets `err` to point it. If `gtk_dialog_save_finish` returns a GFile, the function `save_file` is called.
- 12-21: If the file is successfully saved, these lines are executed. See the comments, line 12-15, for the details.
- 23-28: If an error happens, show the error message through the alert dialog.

13.3.3 tfe_text_view_save function

```
1  void
2  tfe_text_view_save (TfeTextView *tv) {
3    g_return_if_fail (TFE_IS_TEXT_VIEW (tv));
4
5    GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
```

```
6     GtkWidget *win = gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_WINDOW);
7
8     if (! gtk_text_buffer_get_modified (tb))
9       return; /* no need to save it */
10    else if (tv->file == NULL)
11      tfe_text_view_saveas (tv);
12    else
13      save_file (tv->file, tb, GTK_WINDOW (win));
14  }
```

- The function `tfe_text_view_save` writes the contents to the `tv->file` file. It calls `tfe_text_view_saveas` or `save_file`.
- 1-3: The function is public, i.e. it is open to the other objects. So, it doesn't have `static` class. Public functions should check the parameter type with `g_return_if_fail` function. If `tv` is not a pointer to a TfeTextView instance, then it logs an error message and immediately returns. This function is similar to `g_return_val_if_fail`, but no value is returned because `tfe_text_view_save` doesn't return a value (void).
- 5-6: GtkTextBuffer `tb` and GtkWidget (GtkWindow) `win` are set. The function `gtk_widget_get_ancestor (widget, type)` returns the first ancestor of the widget with the type, which is a GType. The parent-child relationship here is the one for widgets, not classes. More precisely, the returned widget's type is the `type` or a descendant object type of the `type`. Be careful, the "descendant object" in the previous sentence is *not* "descendant widget". For example, the type of GtkWindow is `GTK_TYPE_WINDOW` and the one of TfeTextView is `TFE_TYPE_TEXT_VIEW`. The top level window may be a GtkApplicationWindow, but it is a descendant of GtkWindow. Therefore, `gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_WINDOW)` possibly returns GtkWindow or GtkApplicationWindow.
- 8-9: If the buffer hasn't modified, it doesn't need to be saved.
- 10-11: If `tv->file` is NULL, which means no file has given yet, it calls `tfe_text_view_saveas` to prompt a user to select a file and save the contents.
- 12-13: Otherwise, it calls `save_file` to save the contents to the file `tv->file`.

13.3.4 tfe_text_view_saveas function

```
1   void
2   tfe_text_view_saveas (TfeTextView *tv) {
3     g_return_if_fail (TFE_IS_TEXT_VIEW (tv));
4
5     GtkWidget *win = gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_WINDOW);
6     GtkFileDialog *dialog;
7
8     dialog = gtk_file_dialog_new ();
9     gtk_file_dialog_save (dialog, GTK_WINDOW (win), NULL, save_dialog_cb, tv);
10    g_object_unref (dialog);
11  }
```

The function `tfe_text_view_saveas` shows a file chooser dialog and prompts the user to choose a file and save the contents.

- 1-3: Check the type of `tv` because the function is public.
- 6: GtkWidget `win` is set to the window which is an ancestor ot `tv`.
- 8: Creates a GtkFileDialog instance. GtkFileDialog is available since version 4.10. If your Gtk version is older than 4.10, use GtkFileChooserDialog instead. GtkFileChooserDialog is deprecated since version 4.10.
- 9: Calls `gtk_file_dialog_save` function. The arguments are:
  - dialog: GtkFileDialog.
  - GTK_WINDOW (win): transient parent window.
  - NULL: NULL means no cancellable object. If you put a cancellable object here, you can cancel the operation by other thread. In many cases, it is NULL. See GCancellable for further information.
  - `save_dialog_cb`: A callback to call when the operation is complete. The type of the pointer to the callback function is GAsyncReadyCallback. If a cancellable object is given and the operation is cancelled, the callback won't be called.

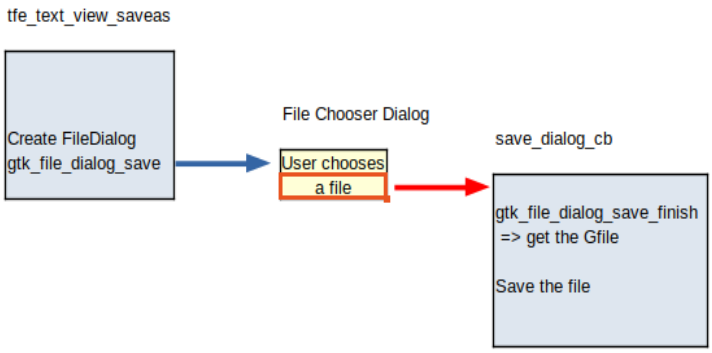


Figure 17: Saveas process

  - `tv`: This is an optional user data which is gpointer type. It is used in the callback function.
- 10: Releases the GtkFileDialog instance because it is useless anymore.

This function just shows the file chooser dialog. The rest of the operation is done by the callback function.

13.4 Open related functions

Open function shows a file chooser dialog to a user and prompts them to choose a file. Then it reads the file and puts the text into GtkTextBuffer.

```
void tfe_text_view_open (TfeTextView *tv, GtkWindow *win);
```

The parameter `win` is the transient window. A file chooser dialog will be shown at the center of the window.

This function may be called just after `tv` has been created. In that case, `tv` has not been incorporated into the widget hierarchy. Therefore it is impossible to get the top-level window from `tv`. That's why the function needs `win` parameter.

This function is usually called when the buffer of `tv` is empty. However, even if the buffer is not empty, `tfe_text_view_open` doesn't treat it as an error. If you want to revert the buffer, calling this function is appropriate.

Open and read process is divided into two phases. One is creating and showing a file chooser dialog and the other is the callback function. The former is `tfe_text_view_open` and the latter is `open_dialog_cb`.

13.4.1 open__dialog__cb function

```
1  static void
2  open_dialog_cb (GObject *source_object, GAsyncResult *res, gpointer data) {
3    GtkFileDialog *dialog = GTK_FILE_DIALOG (source_object);
4    TfeTextView *tv = TFE_TEXT_VIEW (data);
5    GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
6    GtkWidget *win = gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_WINDOW);
7    GFile *file;
8    char *contents;
9    gsize length;
10   gboolean file_changed;
11   GtkAlertDialog *alert_dialog;
12   GError *err = NULL;
13
14   if ((file = gtk_file_dialog_open_finish (dialog, res, &err)) != NULL
15       && g_file_load_contents (file, NULL, &contents, &length, NULL, &err)) {
16     gtk_text_buffer_set_text (tb, contents, length);
17     g_free (contents);
18     gtk_text_buffer_set_modified (tb, FALSE);
19     // G_IS_FILE(tv->file) && tv->file == file => unref(tv->file), tv->file=file,
         emit response with SUCCESS
```

```
20      // G_IS_FILE(tv->file) && tv->file != file => unref(tv->file), tv->file=file,
            emit response with SUCCESS, emit change-file
21      // tv->file==NULL =>                                          tv->file=file,
            emit response with SUCCESS, emit change-file
22      // The order is important. If you unref tv->file first, you can't compare
            tv->file and file anymore.
23      // And the signals are emitted after new tv->file is set. Or the handler can't
            catch the new file.
24      file_changed = (G_IS_FILE (tv->file) && g_file_equal (tv->file, file)) ? FALSE :
            TRUE;
25      if (G_IS_FILE (tv->file))
26        g_object_unref (tv->file);
27      tv->file = file; // The ownership of 'file' moves to TfeTextView
28      if (file_changed)
29        g_signal_emit (tv, tfe_text_view_signals[CHANGE_FILE], 0);
30      g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0,
            TFE_OPEN_RESPONSE_SUCCESS);
31    } else {
32      if (err->code == GTK_DIALOG_ERROR_DISMISSED) // The user canceled the file
            chooser dialog
33        g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0,
              TFE_OPEN_RESPONSE_CANCEL);
34      else {
35        alert_dialog = gtk_alert_dialog_new ("%s", err->message);
36        gtk_alert_dialog_show (alert_dialog, GTK_WINDOW (win));
37        g_object_unref (alert_dialog);
38        g_signal_emit (tv, tfe_text_view_signals[OPEN_RESPONSE], 0,
              TFE_OPEN_RESPONSE_ERROR);
39      }
40      g_clear_error (&err);
41    }
42  }
```

This function is similar to `save_dialog_cb`. Both are callback functions on a GtkFileDialog object.

- 2: It has three parameters like `save_dialog_cb`. They are:
  - GObject *source_object: The GObject instance that the operation was started with. It is actually the GtkFileDialog instance that is shown to the user. It will be casted to GtkFileDialog later.
  - GAsyncResult *res: The result of the asynchronous operation. It will be given to the `gtk_dialog_open_finish` function.
  - gpointer data: A user data set in the `gtk_dialog_open` function. It is actually a TfeTextView instance and it will be casted to TfeTextView later.
- 14: The function `gtk_file_dialog_open_finish` returns a GFile object if the operation has succeeded. Otherwise it returns NULL.
- 16-30: If the user selects a file and the file has successfully been read, the codes from 16 to 30 will be executed.
- 16-18: Sets the buffer of `tv` with the text read from the file. And frees `contents`. Then sets the modified status to false.
- 19-30: The codes are a bit complicated. See the comments. If the file (`tv->file`) is changed, "change-file" signal is emitted. The signal "open-response" is emitted with the parameter `TFE_OPEN_RESPONSE_SUCCESS`.
- 31-41: If the operation failed, the codes from 31 to 41 will be executed.
- 32-33: If the error code is `GTK_DIALOG_ERROR_DISMISSED`, it means that the user has clicked on the "Cancel" button or close button on the header bar. Then, "open-response" signal is emitted with the parameter `TFE_OPEN_RESPONSE_CANCEL`. The Dialog error is described here in the GTK API reference.
- 35-38: If another error occurs, it shows an alert dialog to report the error and emits "open-response" signal with the parameter `TFE_OPEN_RESPONSE_ERROR`.
- 40: Clears the error structure.

13.4.2 tfe_text_view_open function

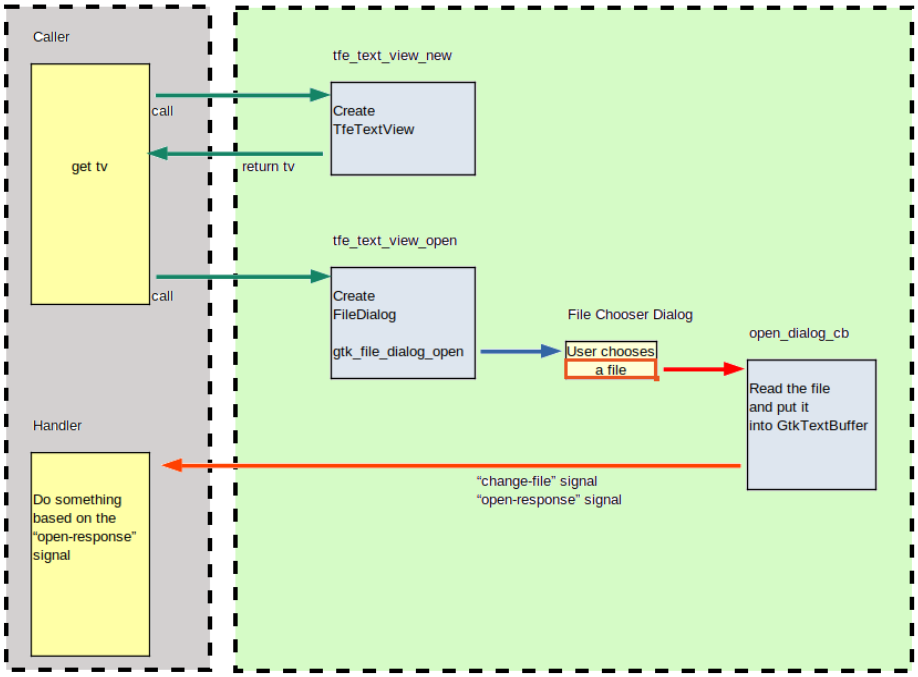


Figure 18: Caller and TfeTextView

```
1  void
2  tfe_text_view_open (TfeTextView *tv, GtkWindow *win) {
3    g_return_if_fail (TFE_IS_TEXT_VIEW (tv));
4    // 'win' is used for a transient window of the GtkFileDialog.
5    // It can be NULL.
6    g_return_if_fail (GTK_IS_WINDOW (win) || win == NULL);
7
8    GtkFileDialog *dialog;
9
10   dialog = gtk_file_dialog_new ();
11   gtk_file_dialog_open (dialog, win, NULL, open_dialog_cb, tv);
12   g_object_unref (dialog);
13 }
```

- 3-6: Check the type of the arguments `tv` and `win`. Public functions always need to check the arguments.
- 10: Creates a GtkFileDialog instance.
- 11: Calls `gtk_file_dialog_open`. The arguments are:
  - `dialog`: the GtkFileDialog instance
  - `win`: the transient window for the file chooser dialog
  - `NULL`: NULL means no cancellable object
  - `open_dialog_cb`: callback function
  - `tv`: user data which is used in the callback function
- 12: Releases the dialog instance because it is useless anymore.

The whole process between the caller and TfeTextView is shown in the following diagram. It is really complicated. Because `gtk_file_dialog_open` can't return the status of the operation.

1. A caller gets a pointer `tv` to a TfeTextView instance by calling `tfe_text_view_new`.
2. The caller connects the handler (left bottom in the diagram) and the signal "open-response".
3. It calls `tfe_text_view_open` to prompt the user to select a file from the file chooser dialog.
4. When the dialog is closed, the callback `open_dialog_cb` is called.
5. The callback function reads the file and inserts the text into GtkTextBuffer and emits a signal to inform the status as a response code.
6. The handler of the "open-response" signal is invoked and the operation status is given to it as an argument (signal parameter).

13.5 Getting GFile in TfeTextView

You can get the GFile in a TfeTextView instance with `tfe_text_view_get_file`. It is very simple.

```
1  GFile *
2  tfe_text_view_get_file (TfeTextView *tv) {
3    g_return_val_if_fail (TFE_IS_TEXT_VIEW (tv), NULL);
4
5    if (G_IS_FILE (tv->file))
6      return g_file_dup (tv->file);
7    else
8      return NULL;
9  }
```

The important thing is to duplicate `tv->file`. Otherwise, if the caller frees the GFile object, `tv->file` is no more guaranteed to point the GFile. Another reason to use `g_file_dup` is that GFile isn't thread-safe. If you use GFile in the different thread, the duplication is necessary. See Gio API Reference – g_file_dup.

13.6 The API document and source file of tfetextview.c

Refer API document of TfeTextView in the appendix. The markdown file is under the directory `src/tfetextview`.

You can find all the TfeTextView source codes under src/tfetextview directories.

14 Functions in GtkNotebook

GtkNotebook is a very important object in the text file editor `tfe`. It connects the application and TfeTextView objects. A set of public functions are declared in `tfenotebook.h`. The word "tfenotebook" is used only in filenames. There's no "TfeNotebook" object.

The source files are in the directory `src/tfe5`. You can get them by downloading the repository.

```
1  void
2  notebook_page_save(GtkNotebook *nb);
3
4  void
5  notebook_page_close (GtkNotebook *nb);
6
7  void
8  notebook_page_open (GtkNotebook *nb);
9
10 void
11 notebook_page_new_with_file (GtkNotebook *nb, GFile *file);
12
13 void
14 notebook_page_new (GtkNotebook *nb);
```

This header file describes the public functions in `tfenotebook.c`.

- 1-2: `notebook_page_save` saves the current page to the file of which the name specified in the tab. If the name is `untitled` or `untitled` followed by digits, a file chooser dialog appears and a user can choose or specify a filename.
- 4-5: `notebook_page_close` closes the current page.
- 7-8: `notebook_page_open` shows a file chooser dialog and a user can choose a file. The contents of the file is inserted to a new page.
- 10-11: `notebook_page_new_with_file` creates a new page and a file given as an argument is read and inserted into the page.
- 13-14: `notebook_page_new` creates a new empty page.

You probably find that the functions except `notebook_page_close` are higher level functions of

- `tfe_text_view_save`
- `tef_text_view_open`

- `tfe_text_view_new_with_file`
- `tfe_text_view_new`

respectively.

There are two layers. One of them is `tfe_text_view ...`, which is the lower level layer. The other is `notebook ...`, which is the higher level layer.

Now let's look at the program of each function.

14.1 notebook_page_new

```
1  static char*
2  get_untitled () {
3    static int c = -1;
4    if (++c == 0)
5      return g_strdup_printf("Untitled");
6    else
7      return g_strdup_printf ("Untitled%u", c);
8  }
9
10 static void
11 notebook_page_build (GtkNotebook *nb, GtkWidget *tv, const char *filename) {
12   GtkWidget *scr = gtk_scrolled_window_new ();
13   GtkNotebookPage *nbp;
14   GtkWidget *lab;
15   int i;
16
17   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
18   lab = gtk_label_new (filename);
19   i = gtk_notebook_append_page (nb, scr, lab);
20   nbp = gtk_notebook_get_page (nb, scr);
21   g_object_set (nbp, "tab-expand", TRUE, NULL);
22   gtk_notebook_set_current_page (nb, i);
23   g_signal_connect (GTK_TEXT_VIEW (tv), "change-file", G_CALLBACK (file_changed_cb),
         nb);
24 }
25
26 void
27 notebook_page_new (GtkNotebook *nb) {
28   g_return_if_fail(GTK_IS_NOTEBOOK (nb));
29
30   GtkWidget *tv;
31   char *filename;
32
33   tv = tfe_text_view_new ();
34   filename = get_untitled ();
35   notebook_page_build (nb, tv, filename);
36   g_free (filename);
37 }
```

- 26-37: The function `notebook_page_new`.
- 28: The function `g_return_if_fail` checks the argument. It's necessary because the function is public.
- 33: Creates TfeTextView object.
- 34: Creates filename, which is "Untitled", "Untitled1", ...
- 1-8: The function `get_untitled`.
- 3: Static variable `c` is initialized at the first call of this function. After that `c` keeps its value unless it is changed explicitly.
- 4-7: Increases `c` by one and if it is zero, it returns "Untitled". If it is a positive integer, it returns "Untitled<the integer>", for example, "Untitled1", "Untitled2", and so on. The function `g_strdup_printf` creates a string and it should be freed by `g_free` when it becomes useless. The caller of `get_untitled` is in charge of freeing the string.
- 36: Calls `notebook_page_build` to build a new page.
- 37: Frees `filename`.

- 10- 24: The function `notebook_page_build`. A parameter with `const` qualifier doesn't change in the function. It means that the argument `filename` is owned by the caller. The caller needs to free it when it becomes useless.
- 12: Creates GtkScrolledWindow.
- 17: Inserts `tv` to GtkScrolledWindow as a child.
- 18-19: Creates GtkLabel, then appends `scr` and `lab` to the GtkNotebook instance `nb`.
- 20-21: Sets "tab-expand" property to TRUE. The function `g_object_set` sets properties on an object. The object can be any object derived from GObject. In many cases, an object has its own function to set its properties, but sometimes doesn't. In that case, use `g_object_set` to set the property.
- 22: Sets the current page to the newly created page.
- 23: Connects "change-file" signal and the handler `file_changed_cb`.

14.2 notebook_page_new_with_file

```
1  void
2  notebook_page_new_with_file (GtkNotebook *nb, GFile *file) {
3    g_return_if_fail(GTK_IS_NOTEBOOK (nb));
4    g_return_if_fail(G_IS_FILE (file));
5
6    GtkWidget *tv;
7    char *filename;
8
9    if ((tv = tfe_text_view_new_with_file (file)) == NULL)
10     return; /* read error */
11   filename = g_file_get_basename (file);
12   notebook_page_build (nb, tv, filename);
13   g_free (filename);
14 }
```

- 9-10: Calls `tfe_text_view_new_with_file`. If the function returns NULL, an error has happened. Then, it does nothing and returns.
- 11-13: Gets the filename, builds a new page and frees `filename`.

14.3 notebook_page_open

```
1  static void
2  open_response_cb (TfeTextView *tv, int response, GtkNotebook *nb) {
3    GFile *file;
4    char *filename;
5
6    if (response != TFE_OPEN_RESPONSE_SUCCESS) {
7      g_object_ref_sink (tv);
8      g_object_unref (tv);
9    }else {
10     file = tfe_text_view_get_file (tv);
11     filename = g_file_get_basename (file);
12     g_object_unref (file);
13     notebook_page_build (nb, GTK_WIDGET (tv), filename);
14     g_free (filename);
15   }
16 }
17
18 void
19 notebook_page_open (GtkNotebook *nb) {
20   g_return_if_fail(GTK_IS_NOTEBOOK (nb));
21
22   GtkWidget *tv;
23
24   tv = tfe_text_view_new ();
25   g_signal_connect (TFE_TEXT_VIEW (tv), "open-response", G_CALLBACK
         (open_response_cb), nb);
26   tfe_text_view_open (TFE_TEXT_VIEW (tv), GTK_WINDOW (gtk_widget_get_ancestor
         (GTK_WIDGET (nb), GTK_TYPE_WINDOW)));
```

```
27 }
```

- 18-27: The function `notebook_page_open`.
- 24: Creates TfeTextView object.
- 25: Connects the signal "open-response" and the handler `open_response_cb`.
- 26: Calls `tfe_text_view_open`. The "open-response" signal will be emitted later in this function to inform the result.
- 1-16: The handler `open_response_cb`.
- 6-8: If the response code is not `TFE_OPEN_RESPONSE_SUCCESS`, the instance `tv` will be destroyed. It has floating reference, which will be explained later. A floating reference needs to be converted into an ordinary reference before releasing it. The function `g_object_ref_sink` does that. After that, the function `g_object_unref` releases `tv` and decreases the reference count by one. Finally the reference count becomes zero and `tv` is destroyed.
- 9-15: Otherwise, it builds a new page with `tv`.

14.4 Floating reference

All the widgets are derived from GInitiallyUnowned. GObject and GInitiallyUnowned are almost the same. The difference is like this. When an instance of GInitiallyUnowned is created, the instance has a "floating reference". On the other hand, when an instance of GObject (not GInitiallyUnowned) is created, it has "normal reference". Their descendants inherits them, so every widget has a floating reference just after the creation. Non-widget class, for example, GtkTextBuffer is a direct sub class of GObject and it has normal reference.

The function `g_object_ref_sink` converts the floating reference into a normal reference. If the instance doesn't have a floating reference, `g_object_ref_sink` simply increases the reference count by one. It is used when an widget is added to another widget as a child.

```
GtkTextView *tv = gtk_text_view_new (); // Floating reference
GtkScrolledWindow *scr = gtk_scrolled_window_new ();
gtk_scrolled_window_set_child (scr, tv); // Scrolled window sink the tv's floating
    reference and tv's reference count becomes one.
```

When `tv` is added to `scr` as a child, `g_object_ref_sink` is used.

```
g_object_ref_sink (tv);
```

So, the floating reference is converted into an ordinary reference. That is to say, floating reference is removed, and the normal reference count is one. Thanks to this, the caller doesn't need to decrease tv's reference count. If an Object_A is not a descendant of GInitiallyUnowned, the program is like this:

```
Object_A *obj_a = object_a_new (); // reference count is one
GtkScrolledWindow *scr = gtk_scrolled_window_new ();
gtk_scrolled_window_set_child (scr, obj_a); // obj_a's reference count is two
// obj_a is referred by the caller (this program) and scrolled window
g_object_unref (obj_a); // obj_a's reference count is one because the caller no
    longer refers obj_a.
```

This example tells us that the caller needs to unref `obj_a`.

If you use `g_object_unref` to an instance that has a floating reference, you need to convert the floating reference to a normal reference in advance. See GObject API reference for further information.

14.5 notebook_page_close

```
1 void
2 notebook_page_close (GtkNotebook *nb) {
3   g_return_if_fail(GTK_IS_NOTEBOOK (nb));
4
5   GtkWidget *win;
6   int i;
7
8   if (gtk_notebook_get_n_pages (nb) == 1) {
```

```
9       win = gtk_widget_get_ancestor (GTK_WIDGET (nb), GTK_TYPE_WINDOW);
10      gtk_window_destroy(GTK_WINDOW (win));
11    } else {
12      i = gtk_notebook_get_current_page (nb);
13      gtk_notebook_remove_page (GTK_NOTEBOOK (nb), i);
14    }
15  }
```

This function closes the current page. If the page is the only page the notebook has, then the function destroys the top-level window and quits the application.

- 8-10: If the page is the only page the notebook has, it calls `gtk_window_destroy` to destroy the top-level window.
- 11-13: Otherwise, removes the current page. The child widget (TfeTextView) is also destroyed.

14.6 notebook_page_save

```
1   static TfeTextView *
2   get_current_textview (GtkNotebook *nb) {
3     int i;
4     GtkWidget *scr;
5     GtkWidget *tv;
6
7     i = gtk_notebook_get_current_page (nb);
8     scr = gtk_notebook_get_nth_page (nb, i);
9     tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
10    return TFE_TEXT_VIEW (tv);
11  }
12
13  void
14  notebook_page_save (GtkNotebook *nb) {
15    g_return_if_fail(GTK_IS_NOTEBOOK (nb));
16
17    TfeTextView *tv;
18
19    tv = get_current_textview (nb);
20    tfe_text_view_save (tv);
21  }
```

- 13-21: `notebook_page_save`.
- 19: Gets the TfeTextView instance belongs to the current page. The caller doesn't have the ownership of `tv` so you don't need to care about freeing it.
- 20: Calls `tfe_text_view_save`.
- 1-11: `get_current_textview`. This function gets the TfeTextView object belongs to the current page.
- 7: Gets the page number of the current page.
- 8: Gets the child widget `scr`, which is a GtkScrolledWindow instance, of the current page. The object `scr` is owned by the notebook `nb`. So, the caller doesn't need to free it.
- 9-10: Gets the child widget of `scr`, which is a TfeTextView instance, and returns it. The returned instance is owned by `scr` and the caller of `get_cuurent_textview` doesn't need to care about freeing it.

14.7 file_changed_cb handler

The function `file_changed_cb` is a handler connected to "change-file" signal. If a file in a TfeTextView instance is changed, the instance emits this signal. This handler changes the label of the GtkNotebookPage.

```
1   static void
2   file_changed_cb (TfeTextView *tv, GtkNotebook *nb) {
3     GtkWidget *scr;
4     GtkWidget *label;
5     GFile *file;
6     char *filename;
7
8     file = tfe_text_view_get_file (tv);
```

```
9    scr = gtk_widget_get_parent (GTK_WIDGET (tv));
10   if (G_IS_FILE (file)) {
11     filename = g_file_get_basename (file);
12     g_object_unref (file);
13   } else
14     filename = get_untitled ();
15   label = gtk_label_new (filename);
16   g_free (filename);
17   gtk_notebook_set_tab_label (nb, scr, label);
18 }
```

- 8: Gets the GFile instance from `tv`.
- 9: Gets the GkScrolledWindow instance which is the parent widget of `tv`.
- 10-12: If `file` points a GFile instance, the filename of the GFile is assigned to `filename`. Then, unref the GFile object `file`.
- 13-14: Otherwise (file is NULL), a string `Untitled(number)` is assigned to `filename`.
- 15-17: Creates a GtkLabel instance `label` with the filename and set the label of the GtkNotebookPage with `label`.

15 Tfe main program

The file `tfeapplication.c` is a main program of Tfe. It includes all the code other than `tfetextview.c` and `tfenotebook.c`. It does:

- Application support, mainly handling command line arguments.
- Builds widgets using ui file.
- Connects button signals and their handlers.
- Manages CSS.

15.1 The function main

The function `main` is the first invoked function in C language. It connects the command line given by the user and Gtk application.

```
1  #define APPLICATION_ID "com.github.ToshioCP.tfe"
2
3  int
4  main (int argc, char **argv) {
5    GtkApplication *app;
6    int stat;
7
8    app = gtk_application_new (APPLICATION_ID, G_APPLICATION_HANDLES_OPEN);
9
10   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
11   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
12   g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
13
14   stat =g_application_run (G_APPLICATION (app), argc, argv);
15   g_object_unref (app);
16   return stat;
17 }
```

- 1: Defines the application id. Thanks to the `#define` directive, it is easy to find the application id.
- 8: Creates GtkApplication object.
- 10-12: Connects “startup”, “activate” and “open” signals to their handlers.
- 14: Runs the application.
- 15-16: Releases the reference to the application and returns the status.

15.2 Startup signal handler

Startup signal is emitted just after the GtkApplication instance is initialized. The handler initializes the whole application which includes not only GtkApplication instance but also widgets and some other objects.

- Builds the widgets using ui file.
- Connects button signals and their handlers.
- Sets CSS.

The handler is as follows.

```
1  static void
2  app_startup (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkBuilder *build;
5    GtkApplicationWindow *win;
6    GtkNotebook *nb;
7    GtkButton *btno;
8    GtkButton *btnn;
9    GtkButton *btns;
10   GtkButton *btnc;
11
12   build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfe/tfe.ui");
13   win = GTK_APPLICATION_WINDOW (gtk_builder_get_object (build, "win"));
14   nb = GTK_NOTEBOOK (gtk_builder_get_object (build, "nb"));
15   gtk_window_set_application (GTK_WINDOW (win), app);
16   btno = GTK_BUTTON (gtk_builder_get_object (build, "btno"));
17   btnn = GTK_BUTTON (gtk_builder_get_object (build, "btnn"));
18   btns = GTK_BUTTON (gtk_builder_get_object (build, "btns"));
19   btnc = GTK_BUTTON (gtk_builder_get_object (build, "btnc"));
20   g_signal_connect_swapped (btno, "clicked", G_CALLBACK (open_cb), nb);
21   g_signal_connect_swapped (btnn, "clicked", G_CALLBACK (new_cb), nb);
22   g_signal_connect_swapped (btns, "clicked", G_CALLBACK (save_cb), nb);
23   g_signal_connect_swapped (btnc, "clicked", G_CALLBACK (close_cb), nb);
24   g_object_unref(build);
25
26 GdkDisplay *display;
27
28   display = gdk_display_get_default ();
29   GtkCssProvider *provider = gtk_css_provider_new ();
30   gtk_css_provider_load_from_data (provider, "textview␣{padding:␣10px;␣font-family:␣
         monospace;␣font-size:␣12pt;}", -1);
31   gtk_style_context_add_provider_for_display (display, GTK_STYLE_PROVIDER
         (provider), GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
32
33   g_signal_connect (win, "destroy", G_CALLBACK (before_destroy), provider);
34   g_object_unref (provider);
35 }
```

- 12-15: Builds widgets using ui resource. Connects the top-level window and the application with `gtk_window_set_application`.
- 16-23: Gets buttons and connects their signals and handlers. The macro `g_signal_connect_swapped` connects a signal and handler like `g_signal_connect`. The difference is that `g_signal_connect_swapped` swaps the user data for the object. For example, the macro on line 20 swaps `nb` for `btno`. So, the handler expects that the first argument is `nb` instead of `btno`.
- 24: Releases the reference to GtkBuilder.
- 26-31: Sets CSS. CSS in Gtk is similar to CSS in HTML. You can set margin, border, padding, color, font and so on with CSS. In this program, CSS is on line 30. It sets padding, font-family and font size of GtkTextView. CSS will be explained in the next subsection.
- 26-28: GdkDisplay is used to set CSS. The default GdkDisplay object can be obtain with the function `gfk_display_get_default`. This function needs to be called after the window creation.
- 33: Connects "destroy" signal on the main window and before_destroy handler. This handler is explained in the next subsection.
- 34: The provider is useless for the startup handler, so it is released. Note: It doesn't mean the destruction of the provider. It is referred by the display so the reference count is not zero.

15.3 CSS in Gtk

CSS is an abbreviation of Cascading Style Sheet. It is originally used with HTML to describe the presentation semantics of a document. You might have found that widgets in Gtk is similar to elements in HTML. It tells that CSS can be applied to Gtk windowing system, too.

15.3.1 CSS nodes, selectors

The syntax of CSS is as follows.

```
selector { color: yellow; padding-top: 10px; ...}
```

Every widget has CSS node. For example, GtkTextView has `textview` node. If you want to set style to GtkTextView, substitute "textview" for `selector` above.

```
textview {color: yellow; ...}
```

Class, ID and some other things can be applied to the selector like Web CSS. Refer to GTK 4 API Reference – CSS in Gtk for further information.

The codes of the startup handler has a CSS string on line 30.

```
textview {padding: 10px; font-family: monospace; font-size: 12pt;}
```

- Padding is a space between the border and contents. This space makes the textview easier to read.
- font-family is a name of font. The font name "monospace" is one of the generic family font keywords.
- Font-size is set to 12pt.

15.3.2 GtkStyleContext, GtkCssProvider and GdkDisplay

GtkStyleContext is deprecated since version 4.10. But two functions `gtk_style_context_add_provider_for_display` and `gtk_style_context_remove_provider_for_display` are not deprecated. They add or remove a css provider object to the GdkDisplay object.

GtkCssProvider is an object which parses CSS for style widgets.

To apply your CSS to widgets, you need to add GtkStyleProvider (the interface of GtkCssProvider) to the GdkDisplay object. You can get the default display object with the function `gdk_display_get_default`. The returned object is owned by the function and you don't have its ownership. So, you don't need to care about releasing it.

Look at the source file of `startup` handler again.

- 28: The display is obtained by `gdk_display_get_default`.
- 29: Creates a GtkCssProvider instance.
- 30: Puts the CSS into the provider. The function `gtk_css_provider_load_from_data` will be deprecated since 4.12 (Not 4.10). The new function `gtk_css_provider_load_from_string` will be used in the future version of Tfe.
- 31: Adds the provider to the display. The last argument of `gtk_style_context_add_provider_for_display` is the priority of the style provider. `GTK_STYLE_PROVIDER_PRIORITY_APPLICATION` is a priority for application-specific style information. Refer to GTK 4 Reference — Constants for more information. You can find other constants, which have "STYLE_PROVIDER_PRIORITY_XXXX" pattern names.

```
1 static void
2 before_destroy (GtkWidget *win, GtkCssProvider *provider) {
3   GdkDisplay *display = gdk_display_get_default ();
4   gtk_style_context_remove_provider_for_display (display, GTK_STYLE_PROVIDER
        (provider));
5 }
```

When a widget is destroyed, or more precisely during its disposing process, a "destroy" signal is emitted. The "before_destroy" handler connects to the signal on the main window. (See the program list of app_startup.) So, it is called when the window is destroyed.

The handler removes the CSS provider from the GdkDisplay.

Note: CSS providers are removed automatically when the application quits. So, even if the handler `before_destroy` is removed, the application works.

15.4 Activate and open signal handler

The handlers of "activate" and "open" signals are `app_activate` and `app_open` respectively. They just create a new GtkNotebookPage.

```
1  static void
2  app_activate (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkWidget *win = GTK_WIDGET (gtk_application_get_active_window (app));
5    GtkWidget *boxv = gtk_window_get_child (GTK_WINDOW (win));
6    GtkNotebook *nb = GTK_NOTEBOOK (gtk_widget_get_last_child (boxv));
7
8    notebook_page_new (nb);
9    gtk_window_present (GTK_WINDOW (win));
10 }
11
12 static void
13 app_open (GApplication *application, GFile ** files, gint n_files, const gchar
       *hint) {
14   GtkApplication *app = GTK_APPLICATION (application);
15   GtkWidget *win = GTK_WIDGET (gtk_application_get_active_window (app));
16   GtkWidget *boxv = gtk_window_get_child (GTK_WINDOW (win));
17   GtkNotebook *nb = GTK_NOTEBOOK (gtk_widget_get_last_child (boxv));
18   int i;
19
20   for (i = 0; i < n_files; i++)
21     notebook_page_new_with_file (nb, files[i]);
22   if (gtk_notebook_get_n_pages (nb) == 0)
23     notebook_page_new (nb);
24   gtk_window_present (GTK_WINDOW (win));
25 }
```

- 1-10: `app_activate`.
- 8-10: Creates a new page and shows the window.
- 12-25: `app_open`.
- 20-21: Creates notebook pages with files.
- 22-23: If no page has created, maybe because of read error, then it creates an empty page.
- 24: Shows the window.

These codes have become really simple thanks to tfenotebook.c and tfetextview.c.

15.5 A primary instance

Only one GApplication instance can be run at a time in a session. The session is a bit difficult concept and also platform-dependent, but roughly speaking, it corresponds to a graphical desktop login. When you use your PC, you probably login first, then your desktop appears until you log off. This is the session.

However, Linux is multi process OS and you can run two or more instances of the same application. Isn't it a contradiction?

When first instance is launched, then it registers itself with its application ID (for example, `com.github.ToshioCP.tfe`). Just after the registration, startup signal is emitted, then activate or open signal is emitted and the instance's main loop runs. I wrote "startup signal is emitted just after the application instance is initialized" in the prior subsection. More precisely, it is emitted after the registration.

If another instance which has the same application ID is launched, it also tries to register itself. Because this is the second instance, the registration of the ID has already done, so it fails. Because of the failure startup signal isn't emitted. After that, activate or open signal is emitted in the primary instance, not on the second instance. The primary instance receives the signal and its handler is invoked. On the other hand, the second instance doesn't receive the signal and it immediately quits.

Try running two instances in a row.

```
$ ./_build/tfe &
[1] 84453
$ ./build/tfe tfeapplication.c
$
```

First, the primary instance opens a window. Then, after the second instance is run, a new notebook page with the contents of `tfeapplication.c` appears in the primary instance's window. This is because the open signal is emitted in the primary instance. The second instance immediately quits so shell prompt soon appears.

15.6 A series of handlers correspond to the button signals

```
1  static void
2  open_cb (GtkNotebook *nb) {
3    notebook_page_open (nb);
4  }
5
6  static void
7  new_cb (GtkNotebook *nb) {
8    notebook_page_new (nb);
9  }
10
11 static void
12 save_cb (GtkNotebook *nb) {
13   notebook_page_save (nb);
14 }
15
16 static void
17 close_cb (GtkNotebook *nb) {
18   notebook_page_close (GTK_NOTEBOOK (nb));
19 }
```

`open_cb`, `new_cb`, `save_cb` and `close_cb` just call corresponding notebook page functions.

15.7 meson.build

```
1  project('tfe', 'c')
2
3  gtkdep = dependency('gtk4')
4
5  gnome=import('gnome')
6  resources = gnome.compile_resources('resources','tfe.gresource.xml')
7
8  sourcefiles=files('tfeapplication.c', 'tfenotebook.c',
       '../tfetextview/tfetextview.c')
9
10 executable('tfe', sourcefiles, resources, dependencies: gtkdep)
```

In this file, just the source file names are modified from the prior version.

15.8 source files

You can download the files from the repository. There are two options.

- Use git and clone.
- Run your browser and open the top page. Then click on "Code" button and click "Download ZIP" in the popup menu. After that, unzip the archive file.

If you use git, run the terminal and type the following.

```
$ git clone https://github.com/ToshioCP/Gtk4-tutorial.git
```

The source files are under /src/tfe5 directory.

16 How to build tfe (text file editor)

16.1 How to compile and execute the text editor 'tfe'.

First, source files are in the Gtk4-tutorila repository. How to download them is written at the end of the previous section.

The following is the instruction of compilation and execution.

- You need meson and ninja.
- If you have installed gtk4 from the source, you need to set environment variables to suit your installation.
- Change your current directory to `src/tfe5` directory.
- Type `meson setup _build` for configuration.
- Type `ninja -C _build` for compilation. Then the application `tfe` is built under the `_build` directory.
- Type `_build/tfe` to execute it.

Then the window appears. There are four buttons, `New`, `Open`, `Save` and `Close`.

- Click on `Open` button, then a file chooser dialog appears. Choose a file in the list and click on `Open` button. Then the file is read and a new Notebook Page appears.
- Edit the file and click on `Save` button, then the text is saved to the original file.
- Click `Close`, then the Notebook Page disappears.
- Click `Close` again, then the `Untitled` Notebook Page disappears and at the same time the application quits.

This is a very simple editor. It is a good practice for you to add more features.

16.2 Total number of lines, words and characters

```
$ LANG=C wc tfe5/meson.build tfe5/tfeapplication.c tfe5/tfe.gresource.xml
    tfe5/tfenotebook.c tfe5/tfenotebook.h tfetextview/tfetextview.c
    tfetextview/tfetextview.h tfe5/tfe.ui
   10    17   294 tfe5/meson.build
  110   334  3601 tfe5/tfeapplication.c
    6     9   153 tfe5/tfe.gresource.xml
  144   390  3668 tfe5/tfenotebook.c
   15    21   241 tfe5/tfenotebook.h
  235   821  8473 tfetextview/tfetextview.c
   32    54   624 tfetextview/tfetextview.h
   61   100  2073 tfe5/tfe.ui
  613  1746 19127 total
```

17 Menus and actions

17.1 Menus

Users often use menus to tell a command to the application. It is like this:

There are two types of objects.

- "File", "Edit", "View", "Cut", "Copy", "Paste" and "Select All". They are called "menu item" or simply "item". When the user clicks one of these items, then something will happen.
- Menubar, submenu referenced by "Edit" item and two sections. They are called "menu". Menu is an ordered list of items. They are similar to arrays.

- Menubar is a menu which has three items, which are "File", "Edit" and "View".
- The menu item labeled "Edit" has a link to the submenu which has two items. These two items don't have labels. Each item refers to a section.
- The first section is a menu which has three items – "Cut", "Copy" and "Paste".
- The second section is a menu which has one item – "Select All".

Menus can build a complicated structure thanks to the links of menu items.

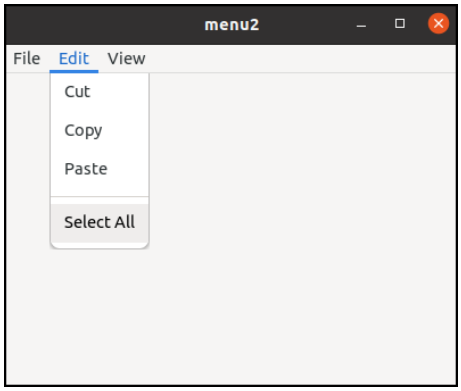


Figure 19: Menu

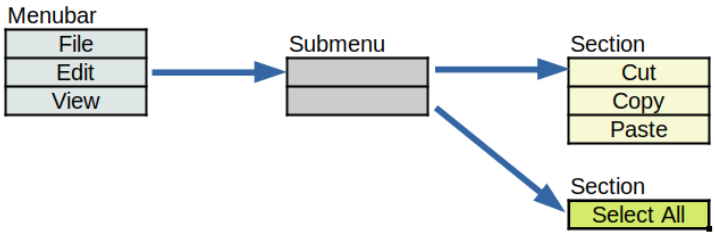


Figure 20: Menu structure

17.2 GMenuModel, GMenu and GMenuItem

GMenuModel is an abstract object which represents a menu. GMenu is a simple implementation of GMenuModel and a child object of GMenuModel.

```
GObject -- GMenuModel -- GMenu
```

Because GMenuModel is an abstract object, it isn't instantiatable. Therefore, it doesn't have any functions to create its instance. If you want to create a menu, use `g_menu_new` to create a GMenu instance. GMenu inherits all the functions of GMenuModel.

GMenuItem is an object directly derived from GObject. GMenuItem and Gmenu (or GMenuModel) don't have a parent-child relationship.

```
GObject -- GMenuModel -- GMenu
GObject -- GMenuItem
```

GMenuItem has attributes. One of the attributes is label. For example, there is a menu item which has "Edit" label in the first diagram. "Cut", "Copy", "Paste" and "Select All" are also the labels of the menu items. Other attributes will be explained later.

Some menu items have a link to another GMenu. There are two types of links, submenu and section.

GMenuItem can be inserted, appended or prepended to GMenu. When it is inserted, all of the attributes and link values are copied and stored in the menu. The GMenuItem itself is not really inserted. Therefore, after the insertion, GMenuItem is useless and it should be freed. The same goes for appending or prepending.

The following code shows how to append GMenuItem to GMenu.

```
GMenu *menu = g_menu_new ();
GMenuItem *menu_item_quit = g_menu_item_new ("Quit", "app.quit");
g_menu_append_item (menu, menu_item_quit);
g_object_unref (menu_item_quit);
```

17.3 Menu and action

One of the attributes of menu items is an action. This attribute points an action object.

There are two action objects, GSimpleAction and GPropertyAction. GSimpleAction is often used. And it is used with a menu item. Only GSimpleAction is described in this section.

An action corresponds to a menu item will be activated when the menu item is clicked. Then the action emits an activate signal.

1. menu item is clicked.
2. The corresponding action is activated.
3. The action emits a signal.
4. The connected handler is invoked.

The following code is an example.

```
static void
quit_activated(GSimpleAction *action, GVariant *parameter, gpointer app) { ... ...
    ...}

GSimpleAction *act_quit = g_simple_action_new ("quit", NULL);
g_action_map_add_action (G_ACTION_MAP (app), G_ACTION (act_quit));
g_signal_connect (act_quit, "activate", G_CALLBACK (quit_activated), app);
GMenuItem *menu_item_quit = g_menu_item_new ("Quit", "app.quit");
```

- The variable `menu_item_quit` points a menu item. It is actually a pointer, but we often say that `menu_item_quit` *is* a menu item. It has a label "Quit" and is connected to an action "app.quit". "app" is a prefix and "quit" is the name of the action. The prefix "app" means that the action belongs to the GtkApplication instance.
- `act_quit` is an action. It has a name "quit". The function `g_simple_action_new` creates a stateless action. So, `act_quit` is stateless. The meaning of stateless will be explained later. The argument `NULL` means that the action doesn't have an parameter. Most of the actions are stateless and have no parameter.
- The action `act_quit` is added to the GtkApplication instance with `g_action_map_add_action`. So, the action's scope is application. The prefix of `app.quit` indicates the scope.
- "activate" signal of the action is connected to the handler `quit_activated`.

If the menu is clicked, the corresponding action "quit" will be activated and emits an "activate" signal. Then, the handler `quit_activated` is called.

17.4 Menu bar

A menu bar and menus are traditional. Menu buttons are often used instead of a menu bar lately, but the old style is still used widely.

Applications have only one menu bar. If an application has two or more windows which have menu bars, the menu bars are exactly the same. Because every window refers to the same menubar instance in the application.

An application's menu bar is usually unchanged once it is set. So, it is appropriate to set it in the "startup" handler. Because the handler is called only once in the primary application instance.

I think it is good for readers to clarify how applications behave.

- When an application runs for the first time, the instance is called primary.
- The primary instance registers itself to the system. If it succeeds, it emits "startup" signal.
- When the instance is activated, an "activate" or "open" signal is emitted.
- If the application is run for the second time or later and there exists a primary instance, the instance is called a remote instance.
- A remote instance doesn't emit "startup" signal.
- If it tries to emit an "activate" or "open" signal, the signals are not emitted on the remote instance but primary instance.
- The remote instance quits.

Therefore, an “activate” or “open” handler can be called twice or more. On the other hand, a “startup” handler is called once. So, the menubar should be set in the “startup” handler.

```
static void
app_startup (GApplication *app) {
... ... ...
  gtk_application_set_menubar (GTK_APPLICATION (app), G_MENU_MODEL (menubar));
... ... ...
}
```

17.5 Simple example

The following is a simple example of menus and actions. The source file `menu1.c` is located at src/menu directory.

```
1  #include <gtk/gtk.h>
2
3  static void
4  quit_activated(GSimpleAction *action, GVariant *parameter, GApplication
       *application) {
5    g_application_quit (application);
6  }
7
8  static void
9  app_activate (GApplication *application) {
10   GtkApplication *app = GTK_APPLICATION (application);
11   GtkWidget *win = gtk_application_window_new (app);
12   gtk_window_set_title (GTK_WINDOW (win), "menu1");
13   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
14
15   gtk_application_window_set_show_menubar (GTK_APPLICATION_WINDOW (win), TRUE);
16   gtk_window_present (GTK_WINDOW (win));
17 }
18
19 static void
20 app_startup (GApplication *application) {
21   GtkApplication *app = GTK_APPLICATION (application);
22
23   GSimpleAction *act_quit = g_simple_action_new ("quit", NULL);
24   g_action_map_add_action (G_ACTION_MAP (app), G_ACTION (act_quit));
25   g_signal_connect (act_quit, "activate", G_CALLBACK (quit_activated), application);
26
27   GMenu *menubar = g_menu_new ();
28   GMenuItem *menu_item_menu = g_menu_item_new ("Menu", NULL);
29   GMenu *menu = g_menu_new ();
30   GMenuItem *menu_item_quit = g_menu_item_new ("Quit", "app.quit");
31   g_menu_append_item (menu, menu_item_quit);
32   g_object_unref (menu_item_quit);
33   g_menu_item_set_submenu (menu_item_menu, G_MENU_MODEL (menu));
34   g_object_unref (menu);
35   g_menu_append_item (menubar, menu_item_menu);
36   g_object_unref (menu_item_menu);
37
38   gtk_application_set_menubar (GTK_APPLICATION (app), G_MENU_MODEL (menubar));
39 }
40
41 #define APPLICATION_ID "com.github.ToshioCP.menu1"
42
43 int
44 main (int argc, char **argv) {
45   GtkApplication *app;
46   int stat;
47
48   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
```

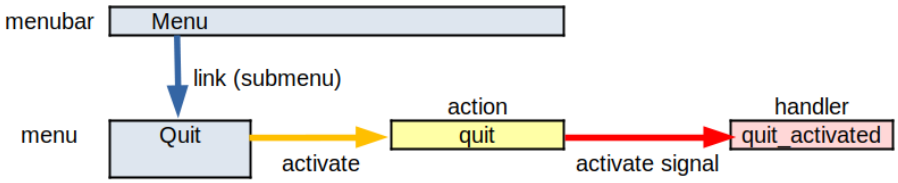


Figure 21: menu and action

```
49    g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
50    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
51
52    stat =g_application_run (G_APPLICATION (app), argc, argv);
53    g_object_unref (app);
54    return stat;
55  }
```

- 3-6: `quit_activated` is a handler of the "activate" signal on the action `act_quit`. Handlers of the "activate" signal have three parameters.
  1. The action instance on which the signal is emitted.
  2. Parameter. In this example it is `NULL` because the second argument of `g_simple_action_new` (line 23) is `NULL`. You don' t need to care about it.
  3. User data. It is the fourth parameter in the `g_signal_connect` (line 25) that connects the action and the handler.
- 5: The function `g_application_quit` immediately quits the application.
- 8-17: `app_activate` is an "activate" signal handler on the application.
- 11-13: Creates a GtkApplicationWindow `win`. And sets the title and the default size.
- 15: Sets GtkApplicationWindow to show the menubar.
- 16: Shows the window.
- 19-38: `app_startup` is a "startup" signal handler on the application.
- 23: Creates GSimpleAction `act_quit`. It is stateless. The first argument of `g_simple_action_new` is a name of the action and the second argument is a parameter. If you don't need the parameter, pass `NULL`. Therefore, `act_quit` has a name "quit" and no parameter.
- 24: Adds the action to GtkApplication `app`. GtkApplication implements an interface GActionMap and GActionGroup. GtkApplication (GActionMap) can have a group of actions and the actions are added with the function `g_action_map_add_action`. This function is described in Gio API Reference – g_action_map_add_action. Because this action belongs to GtkApplication, its scope is "app" and it is referred with "app.quit" if the prefix (scope) is necessary.
- 25: Connects "activate" signal of the action and the handler `quit_activated`.
- 27-30: Creates GMenu and GMenuItem instances. `menubar` and `menu` are GMenu. `menu_item_menu` and `menu_item_quit` are GMenuItem. `menu_item_menu` has a label "Menu" and no action. `menu_item_quit` has a label "Quit" and an action "app.quit".
- 31-32: Appends `menu_item_quit` to `menu`. As I mentioned before, all the attributes and links are copied and used to form a new item in `menu`. Therefore after the addition, `menu_item_quit` is no longer needed. It is freed by `g_object_unref`.
- 33-34: Sets the submenu link in `menu_item_menu` to point `menu`. Then, `menu` is no more useful and it is freed.
- 35-36: Appends `menu_item_menu` to `menubar`. Then frees `menu_item_menu`. GMenu and GMenuItem are built and finally connected to the variable `menubar`. The structure of the menu is shown in the diagram below.
- 38: The menubar is inserted to the application.

17.6 Compiling

Change your current directory to `src/menu`. Use comp to compile `menu1.c`.

```
$ comp menu1
$ ./a.out
```

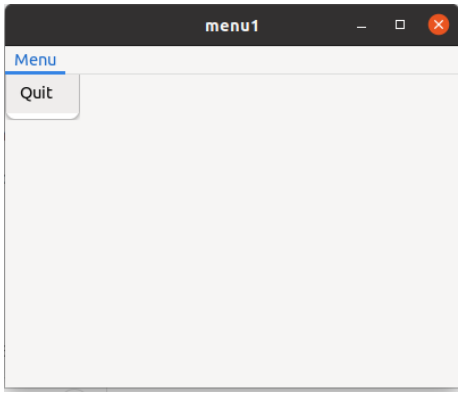


Figure 22: Screenshot of menu1

Then, a window appears. Click on "Menu" on the menubar, then a menu appears. Click on "Quit" menu, then the application quits.

17.7 Primary and remote application instances

Let's try running the application twice. Use `&` in your shell command line, then the application runs concurrently.

```
$ ./a.out &
[1] 70969
$ ./a.out
$
```

Then, two windows appear.

- The first `./a.out` calls the application and a primary instance is created. It calls "startup" and "activate" handlers and shows a window.
- The second `./a.out` calls the the application again and the created instance is a remote one. It doesn't emit "startup" signal. And it activates the application but the "activate" signal is emitted on the primary instance. The remote instance quits.
- The primary instance called "activate" handler. The handler creates a new window. It adds a menu bar to the window with `gtk_application_window_set_show_menubar` function.

Both the windows have menu bars. And they are exactly the same. The two windows belong to the primary instance.

If you click on the "Quit" menu, the application (the primary instance) quits.

The second execution makes a new window. However, it depends on the "activate" handler. If you create your window in the startup handler and the activate handler just presents the window, no new window is created at the second execution. For example, tfe (text file editor) doesn't create a second window. It just creates a new notebook page. Because its activate handler doesn't create any window but just creates a new notebook page.

Second or more executions often happen on the desktop applications. If you double-click the icon twice or more, the application is run multiple times. Therefore, you need to think about your startup and activate (open) handler carefully.

18 Stateful action

Some actions have states. The typical values of states are boolean or string. However, other types of states are possible if you want.

Actions which have states are called stateful.

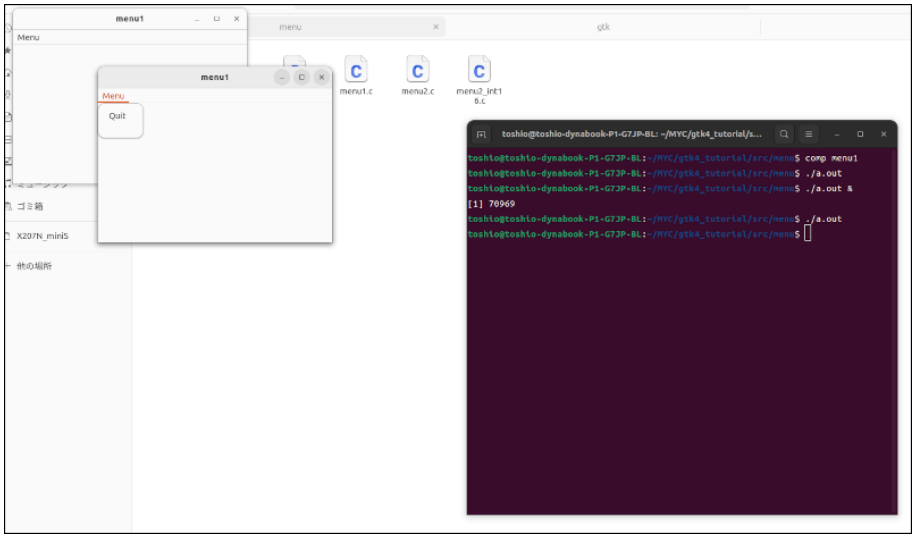


Figure 23: menu1 – two windows

18.1 Stateful action without a parameter

Some menus are called toggle menu. For example, fullscreen menu has a state which has two values – fullscreen and non-fullscreen. The value of the state is changed every time the menu is clicked. An action corresponds to the fullscreen menu also have a state. Its value is TRUE or FALSE and it is called boolean value. TRUE corresponds to fullscreen and FALSE to non-fullscreen.

The following is an example code to implement a fullscreen menu except the signal handler. The signal handler will be shown later.

```
GSimpleAction *act_fullscreen = g_simple_action_new_stateful ("fullscreen",
                                NULL, g_variant_new_boolean (FALSE));
g_signal_connect (act_fullscreen, "change-state", G_CALLBACK (fullscreen_changed),
    win);
g_action_map_add_action (G_ACTION_MAP (win), G_ACTION (act_fullscreen));
... ... ...
GMenuItem *menu_item_fullscreen = g_menu_item_new ("Full␣Screen", "win.fullscreen");
```

- `act_fullscreen` is a GSimpleAction instance. It is created with `g_simple_action_new_stateful`. The function has three arguments. The first argument "fullscreen" is the name of the action. The second argument is a parameter type. `NULL` means the action doesn't have a parameter. The third argument is the initial state of the action. It is a GVariant value. GVariant will be explained in the next subsection. The function `g_variant_new_boolean (FALSE)` returns a boolean type GVariant value which is `FALSE`. If there are two or more top level windows, each window has its own `act_fullscreen` action. So, the number of the actions is the same as the number of the windows.
- The action `act_fullscreen` has "change-state" signal. The signal is connected to a handler `fullscreen_changed`. If the fullscreen menu is clicked, then the corresponding action `act_fullscreen` is activated. But no handler is connected to the "activate" signal. Then, the default behavior for boolean-stated actions with a NULL parameter type like `act_fullscreen` is to toggle them via the "change-state" signal.
- The action is added to the GtkWindow `win`. Therefore, the scope of the action is "win".
- `menu_item_fullscreen` is a GMenuItem instance. There are two arguments. The first argument "Full Screen" is the label of `menu_item_fullscreen`. The second argument is an action. The action "win.fullscreen" has a prefix "win" and an action name "fullscreen". The prefix says that the action belongs to the window.

```
1 static void
2 fullscreen_changed(GSimpleAction *action, GVariant *value, GtkWindow *win) {
3   if (g_variant_get_boolean (value))
4     gtk_window_maximize (win);
5   else
6     gtk_window_unmaximize (win);
```

```
7     g_simple_action_set_state (action, value);
8   }
```

- The handler `fullscreen_changed` has three parameters. The first parameter is the action which emits the "change-state" signal. The second parameter is the value of the new state of the action. The third parameter is a user data which is set in `g_signal_connect`.
- If the value is boolean type and `TRUE`, then it maximizes the window. Otherwise it un-maximizes.
- Sets the state of the action with `value`. Note: At this stage, that means the stage before `g_simple_action_set_state` is called, the state of the action still has the original value. So, you need to set the state with the new value by `g_simple_action_set_state`.

You can use "activate" signal instead of "change-state" signal, or both signals. But the way above is the simplest and the best.

18.1.1 GVariant

GVariant is a fundamental type. It isn't a child of GObject. GVariant can contain boolean, string or other type values. For example, the following program assigns TRUE to `value` whose type is GVariant.

```
GVariant *value = g_variant_new_boolean (TRUE);
```

Another example is:

```
GVariant *value2 = g_variant_new_string ("Hello");
```

`value2` is a GVariant and it has a string type value "Hello". GVariant can contain other types like int16, int32, int64, double and so on.

If you want to get the original value, use g_variant_get series functions. For example, you can get the boolean value with g_variant_get_boolean.

```
gboolean bool = g_variant_get_boolean (value);
```

Since `value` has been created as a boolean type GVariant with `TRUE` value, `bool` equals `TRUE`. In the same way, you can get a string from `value2`

```
const char *str = g_variant_get_string (value2, NULL);
```

The second parameter is a pointer to gsize type variable (gsize is defined as unsigned long). If it isn't NULL, then the pointed value is used as the length by the function. If it is NULL, nothing happens. The returned string `str` is owned by the instance and can't be changed or freed by the caller.

18.2 Stateful action with a parameter

Another example of stateful actions is an action corresponds to color select menus. For example, there are three menus and each menu has red, green or blue color respectively. They determine the background color of a GtkLabel widget. One action is connected to the three menus. The action has a state whose value is "red", "green" or "blue". The values are string. Those colors are given to the signal handler as a parameter.

```
... ... ...
GVariantType *vtype = g_variant_type_new("s");
GSimpleAction *act_color
  = g_simple_action_new_stateful ("color", vtype, g_variant_new_string ("red"));
g_variant_type_free (vtype);
GMenuItem *menu_item_red = g_menu_item_new ("Red", "app.color::red");
GMenuItem *menu_item_green = g_menu_item_new ("Green", "app.color::green");
GMenuItem *menu_item_blue = g_menu_item_new ("Blue", "app.color::blue");
g_signal_connect (act_color, "activate", G_CALLBACK (color_activated), NULL);
... ... ...
```

- GVariantType is a C structure and it keeps a type of GVariant. It is created with the function `g_variant_type_new`. The argument of the function is a GVariant type string. So, `g_variant_type_new("s")` returns a GVariantType structure contains a string type. The returned value, GVariantType structure, is owned by the caller. So, you need to free it when it becomes useless.

- The variable `act_color` points a GSimpleAction instance. It is created with `g_simple_action_new_stateful`. The function has three arguments. The first argument "color" is the name of the action. The second argument is a parameter type which is GVariantType. `g_variant_type_new("s")` creates GVariantType which is a string type (`G_VARIANT_TYPE_STRING`). The third argument is the initial state of the action. It is a GVariant. The function `g_variant_new_string ("red")` returns a GVariant value which has the string value "red". GVariant has a reference count and `g_variant_new_...` series functions returns a GVariant value with a floating reference. That means the caller doesn't own the value at this point. And `g_simple_action_new_stateful` function consumes the floating reference so you don't need to care about releasing the GVariant instance.
- The GVariantType structure `vtype` is useless after `g_simple_action_new_stateful`. It is released with the function `g_variant_type_free`.
- The varable `menu_item_red` points a GMenuItem instance. The function `g_menu_item_new` has two arguments. The first argument "Red" is the label of `menu_item_red`. The second argument is a detailed action. Its prefix is "app", action name is "color" and target is "red". Target is sent to the action as a parameter. The same goes for `menu_item_green` and `menu_item_blue`.
- The function `g_signal_connect` connects the activate signal on the action `act_color` and the handler `color_activated`. If one of the three menus is clicked, then the action `act_color` is activated with the target (parameter) which is given by the menu.

The following is the "activate" signal handler.

```
static void
color_activated(GSimpleAction *action, GVariant *parameter) {
  char *color = g_strdup_printf ("label.lb␣{background-color:␣%s;}",
                                 g_variant_get_string (parameter, NULL));
  gtk_css_provider_load_from_data (provider, color, -1);
  g_free (color);
  g_action_change_state (G_ACTION (action), parameter);
}
```

- The handler originally has three parameters. The third parameter is a user data set in the `g_signal_connect` function. But it is left out because the fourth argument of the `g_signal_connect` has been NULL. The first parameter is the action which emits the "activate" signal. The second parameter is the parameter, or target, given to the action. It is a color specified by the menu.
- The variable `color` is a CSS string created by `g_strdup_printf`. The arguments of `g_strdup_printf` are the same as printf C standard function. The function `g_variant_get_string` gets the string contained in `parameter`. The string is owned by the instance and you mustn't change or free it. The string `label.lb` is a selector. It consists of `label`, a node name of GtkLabel, and `lb` which is a class name. It selects GtkLabel which has `lb` class. For example, menus have GtkLabel to display their labels, but they don't have `lb` class. So, the CSS doesn't change their background color. The string `{background-color %s}` makes the background color `%s` to which the color from `parameter` is assigned.
- Frees the string `color`.
- Changes the state with `g_action_change_state`.

Note: If you haven't set an "activate" signal handler, the signal is forwarded to "change-state" signal. So, you can use "change-state" signal instead of "activate" signal. See `src/menu/menu2_change_state.c`.

18.2.1 GVariantType

GVariantType gives a type of GVariant. GVariantType is created with a type string.

- "b" means boolean type.
- "s" means string type.

The following program is a simple example. It finally outputs the string "s".

```
1  #include <glib.h>
2
3  int
4  main (int argc, char **argv) {
5    GVariantType *vtype = g_variant_type_new ("s");
6    const char *type_string = g_variant_type_peek_string (vtype);
7    g_print ("%s\n",type_string);
8    g_variant_type_free (vtype);
```

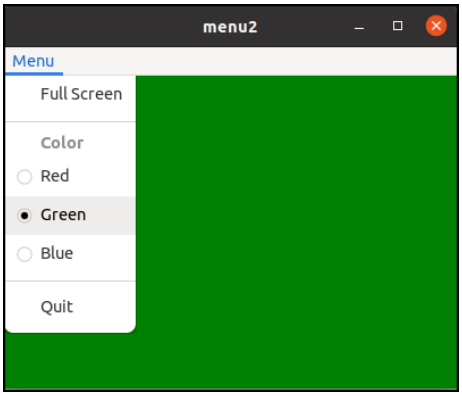


Figure 24: menu2

```
}
```

- The function `g_variant_type_new` creates a GVariantType structure. The argument "s" is a type string. It means string. The returned structure is owned by the caller. When it becomes useless, you need to free it with the function `g_variant_type_free`.
- The function `g_variant_type_peek_string` takes a peek at `vtype`. It is the string "s" given to `vtype` when it was created. The string is owned by the instance and the caller can't change or free it.
- Prints the string to the terminal. You can't free `vtype` before `g_print` because the string `type_string` is owned by `vtype`.
- Frees `vtype`.

18.3 Example

The following code includes stateful actions above. This program has menus like this:

- Fullscreen menu toggles the size of the window between maximum and non-maximum. If the window is maximum size, which is called full screen, then a check mark is put before "fullscreen" label.
- Red, green and blue menu determines the back ground color of the label in the window. The menus have radio buttons on the left of the menus. And the radio button of the selected menu turns on.
- Quit menu quits the application.

The code is as follows.

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

/* The provider below provides application wide CSS data. */
GtkCssProvider *provider;

static void
fullscreen_changed(GSimpleAction *action, GVariant *value, GtkWindow *win) {
  if (g_variant_get_boolean (value))
    gtk_window_maximize (win);
  else
    gtk_window_unmaximize (win);
  g_simple_action_set_state (action, value);
}

static void
color_activated(GSimpleAction *action, GVariant *parameter) {
  char *color = g_strdup_printf ("label.lb {background-color: %s;}",
      g_variant_get_string (parameter, NULL));
  /* Change the CSS data in the provider. */
  /* Previous data is thrown away. */
  gtk_css_provider_load_from_data (provider, color, -1);
  g_free (color);
  g_action_change_state (G_ACTION (action), parameter);
}
```

```
24
25  static void
26  app_shutdown (GApplication *app, GtkCssProvider *provider) {
27    gtk_style_context_remove_provider_for_display (gdk_display_get_default(),
        GTK_STYLE_PROVIDER (provider));
28  }
29
30  static void
31  app_activate (GApplication *app) {
32    GtkWindow *win = GTK_WINDOW (gtk_application_window_new (GTK_APPLICATION (app)));
33    gtk_window_set_title (win, "menu2");
34    gtk_window_set_default_size (win, 400, 300);
35
36    GtkWidget *lb = gtk_label_new (NULL);
37    gtk_widget_add_css_class (lb, "lb"); /* the class is used by CSS Selector */
38    gtk_window_set_child (win, lb);
39
40    GSimpleAction *act_fullscreen
41      = g_simple_action_new_stateful ("fullscreen", NULL, g_variant_new_boolean
          (FALSE));
42    g_signal_connect (act_fullscreen, "change-state", G_CALLBACK (fullscreen_changed),
        win);
43    g_action_map_add_action (G_ACTION_MAP (win), G_ACTION (act_fullscreen));
44
45    gtk_application_window_set_show_menubar (GTK_APPLICATION_WINDOW (win), TRUE);
46
47    gtk_window_present (win);
48  }
49
50  static void
51  app_startup (GApplication *app) {
52    GVariantType *vtype = g_variant_type_new("s");
53    GSimpleAction *act_color
54      = g_simple_action_new_stateful ("color", vtype, g_variant_new_string ("red"));
55    g_variant_type_free (vtype);
56    GSimpleAction *act_quit
57      = g_simple_action_new ("quit", NULL);
58    g_signal_connect (act_color, "activate", G_CALLBACK (color_activated), NULL);
59    g_signal_connect_swapped (act_quit, "activate", G_CALLBACK (g_application_quit),
        app);
60    g_action_map_add_action (G_ACTION_MAP (app), G_ACTION (act_color));
61    g_action_map_add_action (G_ACTION_MAP (app), G_ACTION (act_quit));
62
63    GMenu *menubar = g_menu_new ();
64    GMenu *menu = g_menu_new ();
65    GMenu *section1 = g_menu_new ();
66    GMenu *section2 = g_menu_new ();
67    GMenu *section3 = g_menu_new ();
68    GMenuItem *menu_item_fullscreen = g_menu_item_new ("Full␣Screen",
        "win.fullscreen");
69    GMenuItem *menu_item_red = g_menu_item_new ("Red", "app.color::red");
70    GMenuItem *menu_item_green = g_menu_item_new ("Green", "app.color::green");
71    GMenuItem *menu_item_blue = g_menu_item_new ("Blue", "app.color::blue");
72    GMenuItem *menu_item_quit = g_menu_item_new ("Quit", "app.quit");
73
74    g_menu_append_item (section1, menu_item_fullscreen);
75    g_menu_append_item (section2, menu_item_red);
76    g_menu_append_item (section2, menu_item_green);
77    g_menu_append_item (section2, menu_item_blue);
78    g_menu_append_item (section3, menu_item_quit);
79    g_object_unref (menu_item_red);
80    g_object_unref (menu_item_green);
81    g_object_unref (menu_item_blue);
82    g_object_unref (menu_item_fullscreen);
```

```
83    g_object_unref (menu_item_quit);
84
85    g_menu_append_section (menu, NULL, G_MENU_MODEL (section1));
86    g_menu_append_section (menu, "Color", G_MENU_MODEL (section2));
87    g_menu_append_section (menu, NULL, G_MENU_MODEL (section3));
88    g_menu_append_submenu (menubar, "Menu", G_MENU_MODEL (menu));
89    g_object_unref (section1);
90    g_object_unref (section2);
91    g_object_unref (section3);
92    g_object_unref (menu);
93
94    gtk_application_set_menubar (GTK_APPLICATION (app), G_MENU_MODEL (menubar));
95
96    provider = gtk_css_provider_new ();
97    /* Initialize the css data */
98    gtk_css_provider_load_from_data (provider, "label.lb␣{background-color:␣red;}",
          -1);
99    /* Add CSS to the default GdkDisplay. */
100   gtk_style_context_add_provider_for_display (gdk_display_get_default (),
101         GTK_STYLE_PROVIDER (provider), GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
102   g_signal_connect (app, "shutdown", G_CALLBACK (app_shutdown), provider);
103   g_object_unref (provider); /* release provider, but it's still alive because the
          display owns it */
104 }
105
106 #define APPLICATION_ID "com.github.ToshioCP.menu2"
107
108 int
109 main (int argc, char **argv) {
110   GtkApplication *app;
111   int stat;
112
113   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
114   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
115   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
116
117   stat =g_application_run (G_APPLICATION (app), argc, argv);
118   g_object_unref (app);
119   return stat;
120 }
```

- 6-23: Action signal handlers.
- 25-28: The handler `app_shutdown` is called when the application quits. It removes the provider from the display.
- 30-48: An activate signal handler.
- 32-34: A new window is created and assigned to `win`. Its title and default size are set to “menu2” and 400x300 respectively.
- 36-38: A new label is created and assigned to `lb` The label is given a CSS class “lb”. It is added to `win` as a child.
- 40-43: A toggle action is created and assigned to `act_fullscreen`. It’s connected to the signal handler `fullscreen_changed`. It’s added to the window, so the action scope is “win”. So, if there are two or more windows, the actions are created two or more.
- 45: The function `gtk_application_window_set_show_menubar` adds a menubar to the window.
- 47: Shows the window.
- 50-104: A startup signal handler.
- 52-61: Two actions `act_color` and `act_quit` are created. These actions exists only one because the startup handler is called once. They are connected to their handlers and added to the application. Their scopes are “app”.
- 63-92: Menus are built.
- 94: The menubar is added to the application.
- 96-103: A CSS provider is created with the CSS data and added to the default display. The “shutdown” signal on the application is connected to a handler “app_shutdown”. So, the provider is removed

from the display and freed when the application quits.

18.4 Compile

Change your current directory to `src/menu`.

```
$ comp menu2
$./a.out
```

Then, you will see a window and the background color of the content is red. You can change the size to maximum and change back to the original size. You can change the background color to green or blue.

If you run the second application during the first application is running, another window will appear in the same screen. Both of the window have the same background color. Because the `act_color` action has "app" scope and the CSS is applied to the default display shared by the windows.

```
$ ./a.out & # Run the first application
[1] 82113
$ ./a.out # Run the second application
$
```

19 Ui file for menu and action entries

19.1 Ui file for menu

You may have thought that building menus was really bothersome. Yes, the program was complicated and it needs lots of time to code them. The situation is similar to building widgets. When we built widgets, using ui file was a good way to avoid such complication. The same goes for menus.

The ui file for menus has interface and menu tags. The file starts and ends with interface tags.

```
<interface>
  <menu id="menubar">
  </menu>
</interface>
```

`menu` tag corresponds to GMenu object. `id` attribute defines the name of the object. It will be referred by GtkBuilder.

```
<submenu>
  <attribute name="label">File</attribute>
    <item>
      <attribute name="label">New</attribute>
      <attribute name="action">win.new</attribute>
    </item>
</submenu>
```

`item` tag corresponds to an item in the GMenu which has the same structure as GMenuItem. The item above has a label attribute. Its value is "New". The item also has an action attribute and its value is "win.new". "win" is a prefix and "new" is an action name. `submenu` tag corresponds to both GMenuItem and GMenu. The GMenuItem has a link to GMenu.

The ui file above can be described as follows.

```
<item>
  <attribute name="label">File</attribute>
    <link name="submenu">
      <item>
        <attribute name="label">New</attribute>
        <attribute name="action">win.new</attribute>
      </item>
    </link>
</item>
```

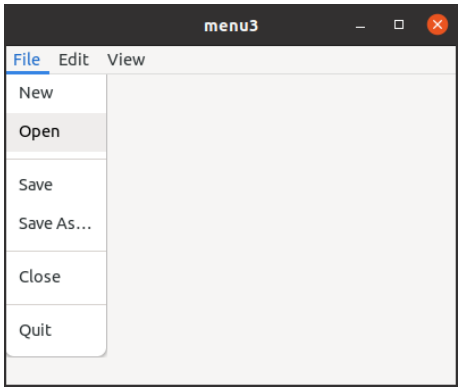


Figure 25: menu3

`link` tag expresses the link to submenu. And at the same time it also expresses the submenu itself. This file illustrates the relationship between the menus and items better than the prior ui file. But `submenu` tag is simple and easy to understand. So, we usually prefer the former ui style.

For further information, see GTK 4 API reference – PopoverMenu.

The following is a screenshot of the sample program `menu3`. It is located in the directory src/menu3.

The following is the ui file for `menu3`.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <menu id="menubar">
    <submenu>
      <attribute name="label">File</attribute>
      <section>
        <item>
          <attribute name="label">New</attribute>
          <attribute name="action">app.new</attribute>
        </item>
        <item>
          <attribute name="label">Open</attribute>
          <attribute name="action">app.open</attribute>
        </item>
      </section>
      <section>
        <item>
          <attribute name="label">Save</attribute>
          <attribute name="action">win.save</attribute>
        </item>
        <item>
          <attribute name="label">Save …As</attribute>
          <attribute name="action">win.saveas</attribute>
        </item>
      </section>
      <section>
        <item>
          <attribute name="label">Close</attribute>
          <attribute name="action">win.close</attribute>
        </item>
      </section>
      <section>
        <item>
          <attribute name="label">Quit</attribute>
          <attribute name="action">app.quit</attribute>
        </item>
      </section>
    </submenu>
```

```
39      <submenu>
40        <attribute name="label">Edit</attribute>
41        <section>
42          <item>
43            <attribute name="label">Cut</attribute>
44            <attribute name="action">app.cut</attribute>
45          </item>
46          <item>
47            <attribute name="label">Copy</attribute>
48            <attribute name="action">app.copy</attribute>
49          </item>
50          <item>
51            <attribute name="label">Paste</attribute>
52            <attribute name="action">app.paste</attribute>
53          </item>
54        </section>
55        <section>
56          <item>
57            <attribute name="label">Select All</attribute>
58            <attribute name="action">app.selectall</attribute>
59          </item>
60        </section>
61      </submenu>
62      <submenu>
63        <attribute name="label">View</attribute>
64        <section>
65          <item>
66            <attribute name="label">Full Screen</attribute>
67            <attribute name="action">win.fullscreen</attribute>
68          </item>
69        </section>
70      </submenu>
71    </menu>
72  </interface>
```

The ui file is converted to the resource by the resource compiler `glib-compile-resouces` with xml file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <gresources>
3    <gresource prefix="/com/github/ToshioCP/menu3">
4      <file>menu3.ui</file>
5    </gresource>
6  </gresources>
```

GtkBuilder builds menus from the resource.

```
GtkBuilder *builder = gtk_builder_new_from_resource
    ("/com/github/ToshioCP/menu3/menu3.ui");
GMenuModel *menubar = G_MENU_MODEL (gtk_builder_get_object (builder, "menubar"));

gtk_application_set_menubar (GTK_APPLICATION (app), menubar);
g_object_unref (builder);
```

The builder instance is freed after the GMenuModel `menubar` is inserted to the application. If you do it before the insertion, bad thing will happen – your computer might freeze. It is because you don't own the `menubar` instance. The function `gtk_builder_get_object` just returns the pointer to `menubar` and doesn't increase the reference count of `menubar`. So, if you released `bulder` before `gtk_application_set_menubar`, `builder` would be destroyed and `menubar` as well.

19.2 Action entry

The coding for building actions and signal handlers is bothersome work as well. Therefore, it should be automated. You can implement them easily with GActionEntry structure and `g_action_map_add_action_entries` function.

GActionEntry contains action name, signal handlers, parameter and state.

```
typedef struct _GActionEntry GActionEntry;

struct _GActionEntry
{
  /* action name */
  const char *name;
  /* activate handler */
  void (* activate) (GSimpleAction *action, GVariant *parameter, gpointer user_data);
  /* the type of the parameter given as a single GVariant type string */
  const char *parameter_type;
  /* initial state given in GVariant text format */
  const char *state;
  /* change-state handler */
  void (* change_state) (GSimpleAction *action, GVariant *value, gpointer user_data);
  /*< private >*/
  gsize padding[3];
};
```

For example, the actions in the previous section are:

```
{ "fullscreen", NULL, NULL, "false", fullscreen_changed }
{ "color", color_activated, "s", "'red'", NULL }
{ "quit", quit_activated, NULL, NULL, NULL },
```

- Fullscreen action is stateful, but doesn't have parameters. So, the third element (parameter type) is NULL. GVariant text format provides "true" and "false" as boolean GVariant values. The initial state of the action is false (the fourth element). It doesn't have activate handler, so the second element is NULL. Instead, it has change-state handler. The fifth element `fullscreen_changed` is the handler.
- Color action is stateful and has a parameter. The parameter type is string. GVariant format strings provides string formats to represent GVariant types. The third element "s" means GVariant string type. GVariant text format defines that strings are surrounded by single or double quotes. So, the string red is 'red' or "red". The fourth element is `"'red'"`, which is a C string format and the string is 'red'. You can write `"\"red\""` instead. The second element `color_activated` is the activate handler. The action doesn't have change-state handler, so the fifth element is NULL.
- Quit action is non-stateful and has no parameter. So, the third and fourth elements are NULL. The second element `quit_activated` is the activate handler. The action doesn't have change-state handler, so the fifth element is NULL.

The function `g_action_map_add_action_entries` does everything to create GSimpleAction instances and add them to a GActionMap (an application or window).

```
const GActionEntry app_entries[] = {
  { "color", color_activated, "s", "'red'", NULL },
  { "quit", quit_activated, NULL, NULL, NULL }
};
g_action_map_add_action_entries (G_ACTION_MAP (app), app_entries,
                                 G_N_ELEMENTS (app_entries), app);
```

The code above does:

- Builds the "color" and "quit" actions
- Connects the action and the "activate" signal handlers (`color_activated` and `quit_activated`).
- Adds the actions to the action map `app`.

The same goes for the other action.

```
const GActionEntry win_entries[] = {
  { "fullscreen", NULL, NULL, "false", fullscreen_changed }
};
g_action_map_add_action_entries (G_ACTION_MAP (win), win_entries,
                                 G_N_ELEMENTS (win_entries), win);
```

The code above does:

- Builds the “fullscreen” action.
- Connects the action and the signal handler `fullscreen_changed`
- Its initial state is set to false.
- Adds the action to the action map `win`.

19.3 Example

Source files are `menu3.c`, `menu3.ui`, `menu3.gresource.xml` and `meson.build`. They are in the directory src/menu3. The following are `menu3.c` and `meson.build`.

```
1  #include <gtk/gtk.h>
2
3  static void
4  new_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
5  }
6
7  static void
8  open_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
9  }
10
11 static void
12 save_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
13 }
14
15 static void
16 saveas_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
17 }
18
19 static void
20 close_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
21   GtkWindow *win = GTK_WINDOW (user_data);
22
23   gtk_window_destroy (win);
24 }
25
26 static void
27 cut_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
28 }
29
30 static void
31 copy_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
32 }
33
34 static void
35 paste_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
36 }
37
38 static void
39 selectall_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data)
       {
40 }
41
42 static void
43 fullscreen_changed (GSimpleAction *action, GVariant *state, gpointer user_data) {
44   GtkWindow *win = GTK_WINDOW (user_data);
45
46   if (g_variant_get_boolean (state))
47     gtk_window_maximize (win);
48   else
49     gtk_window_unmaximize (win);
50   g_simple_action_set_state (action, state);
51 }
52
53 static void
```

```
54 quit_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
55   GApplication *app = G_APPLICATION (user_data);
56 
57   g_application_quit (app);
58 }
59 
60 static void
61 app_activate (GApplication *app) {
62   GtkWidget *win = gtk_application_window_new (GTK_APPLICATION (app));
63 
64   const GActionEntry win_entries[] = {
65     { "save", save_activated, NULL, NULL, NULL },
66     { "saveas", saveas_activated, NULL, NULL, NULL },
67     { "close", close_activated, NULL, NULL, NULL },
68     { "fullscreen", NULL, NULL, "false", fullscreen_changed }
69   };
70   g_action_map_add_action_entries (G_ACTION_MAP (win), win_entries, G_N_ELEMENTS
       (win_entries), win);
71 
72   gtk_application_window_set_show_menubar (GTK_APPLICATION_WINDOW (win), TRUE);
73 
74   gtk_window_set_title (GTK_WINDOW (win), "menu3");
75   gtk_window_set_default_size (GTK_WINDOW (win), 400, 300);
76   gtk_window_present (GTK_WINDOW (win));
77 }
78 
79 static void
80 app_startup (GApplication *app) {
81   GtkBuilder *builder = gtk_builder_new_from_resource
       ("/com/github/ToshioCP/menu3/menu3.ui");
82   GMenuModel *menubar = G_MENU_MODEL (gtk_builder_get_object (builder, "menubar"));
83 
84   gtk_application_set_menubar (GTK_APPLICATION (app), menubar);
85   g_object_unref (builder);
86 
87   const GActionEntry app_entries[] = {
88     { "new", new_activated, NULL, NULL, NULL },
89     { "open", open_activated, NULL, NULL, NULL },
90     { "cut", cut_activated, NULL, NULL, NULL },
91     { "copy", copy_activated, NULL, NULL, NULL },
92     { "paste", paste_activated, NULL, NULL, NULL },
93     { "selectall", selectall_activated, NULL, NULL, NULL },
94     { "quit", quit_activated, NULL, NULL, NULL }
95   };
96   g_action_map_add_action_entries (G_ACTION_MAP (app), app_entries, G_N_ELEMENTS
       (app_entries), app);
97 }
98 
99 #define APPLICATION_ID "com.github.ToshioCP.menu3"
100 
101 int
102 main (int argc, char **argv) {
103   GtkApplication *app;
104   int stat;
105 
106   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
107   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
108   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
109 
110   stat =g_application_run (G_APPLICATION (app), argc, argv);
111   g_object_unref (app);
112   return stat;
113 }
```

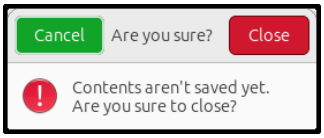


Figure 26: Alert dialog

meson.build

```
project('menu3', 'c')

gtkdep = dependency('gtk4')

gnome=import('gnome')
resources = gnome.compile_resources('resources','menu3.gresource.xml')

sourcefiles=files('menu3.c')

executable('menu3', sourcefiles, resources, dependencies: gtkdep)
```

Action handlers need to follow the following format.

```
static void
handler (GSimpleAction *action_name, GVariant *parameter, gpointer user_data) { ...
    ... ... }
```

You can't write, for example, "GApplication *app" instead of "gpointer user_data". Because `g_action_map_add_action_entries` expects that handlers follow the format above.

There are `menu2_ui.c` and `menu2.ui` under the `menu` directory. They are other examples to show menu ui file and `g_action_map_add_action_entries`. It includes a stateful action with parameters.

```
<item>
  <attribute name="label">Red</attribute>
  <attribute name="action">app.color</attribute>
  <attribute name="target">red</attribute>
</item>
```

Action name and target are separated like this. Action attribute includes prefix and name only. You can't write like `<attribute name="action">app.color::red</attribute>`.

20 Composite widgets and alert dialog

The source files are in the Gtk4 tutorial GitHub repository. Download it and see `src/tfe6` directory.

20.1 An outline of new Tfe text editor

Tfe text editor will be restructured. The program is divided into six parts.

- Main program: the C main function.
- TfeApplication object: It is like GtkApplication but keeps GSettings and CSS Provider.
- TfeWindow object: It is a window with buttons and a notebook.
- TfePref object: A preference dialog.
- TfeAlert object: An alert dialog.
- pdf2css.h and pdf2css.c: Font and CSS utility functions.

This section describes TfeAlert. Others will be explained in the following sections.

20.2 Composite widgets

The alert dialog is like this:

Tfe uses it when a user quits the application or closes a notebook without saving data to files.

The dialog has a title, buttons, an icon and a message. Therefore, it consists of several widgets. Such dialog is called a composite widget.

Composite widgets are defined with template XMLs. The class is built in the class initialization function and the instances are built and desposed by the following functions.

- gtk_widget_init_template
- gtk_widget_dispose_template

TfeAlert is a good example to know composite widgets. It is defined with the three files.

- tfealert.ui: XML file
- tfealert.h: Header file
- tfealert.c: C program file

20.3 The XML file

A template tag is used in a composite widget XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="TfeAlert" parent="GtkWindow">
    <property name="resizable">FALSE</property>
    <property name="modal">TRUE</property>
    <property name="titlebar">
      <object class="GtkHeaderBar">
        <property name="show-title-buttons">FALSE</property>
        <property name="title-widget">
          <object class="GtkLabel" id="lb_title">
            <property name="label">Are you sure?</property>
            <property name="single-line-mode">True</property>
          </object>
        </property>
        <child type="start">
          <object class="GtkButton" id="btn_cancel">
            <property name="label">Cancel</property>
            <style>
              <class name="suggested-action"/>
            </style>
            <signal name="clicked" handler="cancel_cb" swapped="TRUE"
                object="TfeAlert"></signal>
          </object>
        </child>
        <child type="end">
          <object class="GtkButton" id="btn_accept">
            <property name="label">Close</property>
            <style>
              <class name="destructive-action"/>
            </style>
            <signal name="clicked" handler="accept_cb" swapped="TRUE"
                object="TfeAlert"></signal>
          </object>
        </child>
      </object>
    </property>
    <child>
      <object class="GtkBox">
        <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
        <property name="spacing">12</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
        <child>
          <object class="GtkImage">
```

Figure 27: dialog-warning icon is like ...

```
45              <property name="icon-name">dialog-warning</property>
46              <property name="icon-size">GTK_ICON_SIZE_LARGE</property>
47            </object>
48          </child>
49          <child>
50            <object class="GtkLabel" id="lb_message">
51            </object>
52          </child>
53        </object>
54      </child>
55    </template>
56  </interface>
```

- 3: A template tag defines a composite widget. The class attribute tells the class name of the composite widget. The parent attribute tells the parent class of the composite widget. So, TfeAlert is a child class of GtkWindow. A parent attribute is an option and you can leave it out. But it is recommended to write it in the template tag.
- 4-6: Its three properties are defined. These properties are inherited from GtkWindow. The titlebar property has a widget for a custom title bar. The typical widget is GtkHeaderBar.
- 8: If the property "show-title-buttons" is TRUE, the title buttons like close, minimize and maximize are shown. Otherwise it is not shown. The TfeAlert object is not resizable. It is closed when either of the two buttons, cancel or accept, is clicked. Therefore the title buttons are not necessary and this property is set to FALSE.
- 9-14: The bar has a title, which is a GtkLabel widget. The default title is "Are you sure?" but it can be replaced by an instance method.
- 15-32: The bar has two buttons, cancel and accept. The cancel button is on the left so the child tag has `type="start"` attribute. The accept button is on the right so the child tag has `type="end"` attribute. The dialog is shown when the user clicked the close button or the quit menu without saving the data. Therefore, it is safer for the user to click on the cancel button of the alert dialog. So, the cancel button has a "suggested-action" CSS class. Ubuntu colors the button green but the color can be blue or other appropriate one defined by the system. In the same way the accept button has a "destructive-action" CSS class and is colored red. Two buttons have signals which are defined by the signal tags.
- 35-54: A horizontal box has an image icon and a label.
- 44-47: The GtkImage widget displays an image. The "icon-name" property is an icon name in the icon theme. The theme depends on your system. You can check it with an icon browser.

```
$ gtk4-icon-browser
```

The "dialog-warning" icon is something like this.

These are made by my hand. The real image on the alert dialog is nicer.

It is possible to define the alert widget as a child of GtkDialog. But GtkDialog is deprecated since GTK version 4.10. And users should use GtkWindow instead of GtkDialog.

20.4 The header file

The header file is similar to the one of TfeTextView.

```
1  #pragma once
2
3  #include <gtk/gtk.h>
4
5  #define TFE_TYPE_ALERT tfe_alert_get_type ()
```

```
6  G_DECLARE_FINAL_TYPE (TfeAlert, tfe_alert, TFE, ALERT, GtkWindow)
7
8  /* "response" signal id */
9  enum TfeAlertResponseType
10 {
11   TFE_ALERT_RESPONSE_ACCEPT,
12   TFE_ALERT_RESPONSE_CANCEL
13 };
14
15 const char *
16 tfe_alert_get_title (TfeAlert *alert);
17
18 const char *
19 tfe_alert_get_message (TfeAlert *alert);
20
21 const char *
22 tfe_alert_get_button_label (TfeAlert *alert);
23
24 void
25 tfe_alert_set_title (TfeAlert *alert, const char *title);
26
27 void
28 tfe_alert_set_message (TfeAlert *alert, const char *message);
29
30 void
31 tfe_alert_set_button_label (TfeAlert *alert, const char *btn_label);
32
33 GtkWidget *
34 tfe_alert_new (void);
35
36 GtkWidget *
37 tfe_alert_new_with_data (const char *title, const char *message, const char*
       btn_label);
```

- 5-6: These two lines are always needed to define a new object. `TFE_TYPE_ALERT` is the type of TfeAlert object and it is a macro expanded into `tfe_alert_get_type ()`. G_DECLARE_FINAL_TYPE macro is expanded into:
  - The declaration of the function `tfe_alert_get_type`
  - `TfeAlert` is defined as a typedef of `struct _TfeAlert`, which is defined in the C file.
  - `TFE_ALERT` and `TFE_IS_ALERT` macro is defined as a cast and type check function.
  - `TfeAlertClass` structure is defined as a final class.
- 8-13: The TfeAlert class has a "response" signal. It has a parameter and the parameter type is defined as a `TfeAlertResponseType` enumerative constant.
- 15-31: Getter and setter methods.
- 33-37: Functions to create a instance. The function `tfe_alert_new_with_data` is a convenience function, which creates an instance and sets data at once.

20.5 The C file

20.5.1 Functions for composite widgets

The following codes are extracted from `tfealert.c`.

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

struct _TfeAlert {
  GtkWindow parent;
  GtkLabel *lb_title;
  GtkLabel *lb_message;
  GtkButton *btn_accept;
  GtkButton *btn_cancel;
};
```

```
G_DEFINE_FINAL_TYPE (TfeAlert, tfe_alert, GTK_TYPE_WINDOW);

static void
cancel_cb (TfeAlert *alert) {
  ... ... ...
}

static void
accept_cb (TfeAlert *alert) {
  ... ... ...
}

static void
tfe_alert_dispose (GObject *gobject) { // gobject is actually a TfeAlert instance.
  gtk_widget_dispose_template (GTK_WIDGET (gobject), TFE_TYPE_ALERT);
  G_OBJECT_CLASS (tfe_alert_parent_class)->dispose (gobject);
}

static void
tfe_alert_init (TfeAlert *alert) {
  gtk_widget_init_template (GTK_WIDGET (alert));
}

static void
tfe_alert_class_init (TfeAlertClass *class) {
  G_OBJECT_CLASS (class)->dispose = tfe_alert_dispose;
  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
      "/com/github/ToshioCP/tfe/tfealert.ui");
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
      lb_title);
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
      lb_message);
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
      btn_accept);
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
      btn_cancel);
  gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), cancel_cb);
  gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), accept_cb);
  ... ... ...
}

GtkWidget *
tfe_alert_new (void) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_ALERT, NULL));
}
```

- The macro `G_DEFINE_FINAL_TYPE` is available since GLib version 2.70. It is used only for a final type class. You can use `G_DEFINE_TYPE` macro instead. They are expanded into:
  - The declaration of the functions `tfe_alert_init` and `tfe_alert_class_init`. They are defined in the following part of the C program.
  - The definition of the variable `tfe_alert_parent_class`.
  - The definition of the function `tfe_alert_get_type`.
- The names of the members of `_TfeAlert`, which are `lb_title`, `lb_message`, `btn_accept` and `btn_cancel`, must be the same as the id attribute in the XML file `tfealert.ui`.
- The function `tfe_alert_class_init` initializes the composite widget class.
  - The function `gtk_widget_class_set_template_from_resource` sets the template of the class. The template is built from the XML resource "tfealert.ui". At this moment no instance is created. It just makes the class recognize the structure of the object. That's why the top level tag is not object but template in the XML file.
  - The function macro `gtk_widget_class_bind_template_child` connects the member of TfeAlert and the object class in the template. So, for example, you can access to `lb_title` GtkLabel instance via `alert->lb_title` where `alert` is an instance of TfeAlert class.
  - The function `gtk_widget_class_bind_template_callback` connects the callback function and the

`handler` attribute of the signal tag in the XML. For example, the "clicked" signal on the cancel button has a handler named "cancel_cb" in the signal tag. And the function `cancel_cb` exists in the C file above. These two are connected so when the signal is emitted the function `cancel_cb` is called. You can add `static` storage class to the callback function thanks to this connection.

- The function `tfe_alert_init` initializes the newly created instance. You need to call `gtk_widget_init_template` to create and initialize the child widgets in the template.
- The function `tfe_alert_despose` releases objects. The function `gtk_widget_despose_template` clears the template children.
- The function `tfe_alert_new` creates the composite widget TfeAlert instance. It creates not only TfeAlert itself but also all the child widgets that the composite widget has.

20.5.2 Other functions

The following is the full codes of `tfealert.c`.

```
1  #include <gtk/gtk.h>
2  #include "tfealert.h"
3
4  struct _TfeAlert {
5    GtkWindow parent;
6    GtkLabel *lb_title;
7    GtkLabel *lb_message;
8    GtkButton *btn_accept;
9    GtkButton *btn_cancel;
10 };
11
12 G_DEFINE_FINAL_TYPE (TfeAlert, tfe_alert, GTK_TYPE_WINDOW);
13
14 enum {
15   RESPONSE,
16   NUMBER_OF_SIGNALS
17 };
18
19 static guint tfe_alert_signals[NUMBER_OF_SIGNALS];
20
21 static void
22 cancel_cb (TfeAlert *alert) {
23   g_signal_emit (alert, tfe_alert_signals[RESPONSE], 0, TFE_ALERT_RESPONSE_CANCEL);
24   gtk_window_destroy (GTK_WINDOW (alert));
25 }
26
27 static void
28 accept_cb (TfeAlert *alert) {
29   g_signal_emit (alert, tfe_alert_signals[RESPONSE], 0, TFE_ALERT_RESPONSE_ACCEPT);
30   gtk_window_destroy (GTK_WINDOW (alert));
31 }
32
33 const char *
34 tfe_alert_get_title (TfeAlert *alert) {
35   return gtk_label_get_text (alert->lb_title);
36 }
37
38 const char *
39 tfe_alert_get_message (TfeAlert *alert) {
40     return gtk_label_get_text (alert->lb_message);
41 }
42
43 const char *
44 tfe_alert_get_button_label (TfeAlert *alert) {
45   return gtk_button_get_label (alert->btn_accept);
46 }
47
48 void
49 tfe_alert_set_title (TfeAlert *alert, const char *title) {
```

```
50    gtk_label_set_text (alert->lb_title, title);
51  }
52
53  void
54  tfe_alert_set_message (TfeAlert *alert, const char *message) {
55    gtk_label_set_text (alert->lb_message, message);
56  }
57
58  void
59  tfe_alert_set_button_label (TfeAlert *alert, const char *btn_label) {
60    gtk_button_set_label (alert->btn_accept, btn_label);
61  }
62
63  static void
64  tfe_alert_dispose (GObject *gobject) { // gobject is actually a TfeAlert instance.
65    gtk_widget_dispose_template (GTK_WIDGET (gobject), TFE_TYPE_ALERT);
66    G_OBJECT_CLASS (tfe_alert_parent_class)->dispose (gobject);
67  }
68
69  static void
70  tfe_alert_init (TfeAlert *alert) {
71    gtk_widget_init_template (GTK_WIDGET (alert));
72  }
73
74  static void
75  tfe_alert_class_init (TfeAlertClass *class) {
76    G_OBJECT_CLASS (class)->dispose = tfe_alert_dispose;
77    gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
          "/com/github/ToshioCP/tfe/tfealert.ui");
78    gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
          lb_title);
79    gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
          lb_message);
80    gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
          btn_accept);
81    gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeAlert,
          btn_cancel);
82    gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), cancel_cb);
83    gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), accept_cb);
84
85    tfe_alert_signals[RESPONSE] = g_signal_new ("response",
86                                  G_TYPE_FROM_CLASS (class),
87                                  G_SIGNAL_RUN_LAST | G_SIGNAL_NO_RECURSE |
                                      G_SIGNAL_NO_HOOKS,
88                                  0 /* class offset */,
89                                  NULL /* accumulator */,
90                                  NULL /* accumulator data */,
91                                  NULL /* C marshaller */,
92                                  G_TYPE_NONE /* return_type */,
93                                  1     /* n_params */,
94                                  G_TYPE_INT
95                                  );
96  }
97
98  GtkWidget *
99  tfe_alert_new (void) {
100   return GTK_WIDGET (g_object_new (TFE_TYPE_ALERT, NULL));
101 }
102
103 GtkWidget *
104 tfe_alert_new_with_data (const char *title, const char *message, const char*
        btn_label) {
105   GtkWidget *alert = tfe_alert_new ();
106   tfe_alert_set_title (TFE_ALERT (alert), title);
```

```
107   tfe_alert_set_message (TFE_ALERT (alert), message);
108   tfe_alert_set_button_label (TFE_ALERT (alert), btn_label);
109   return alert;
110 }
```

The function `tfe_alert_new_with_data` is used more often than `tfe_alert_new` to create a new instance. It creates the instance and sets three data at the same time. The following is the common process when you use the TfeAlert class.

- Call `tfe_alert_new_with_data` and create an instance.
- Call `gtk_window_set_transient_for` to set the transient parent window.
- Call `gtk_window_present` to show the TfeAlert dialog.
- Connect "response" signal and a handler.
- The user clicks on the cancel or accept button. Then the dialog emits the "response" signal and destroy itself.
- The user catches the signal and do something.

The rest of the program is:

- 14-19: An array for a signal id. You can use a variable instead of an array because the class has only one signal. But using an array is a common way.
- 21-31: Signal handlers. They emits the "response" signal and destroy the instance itself.
- 33-61: Getters and setters.
- 85-95: Creates the "response" signal.
- 103-110: A convenience function `tfe_alert_new_with_data` creates an instance and sets labels.

20.6 An example

There's an example in the `src/tfe6/example` directory. It shows how to use TfeAlert. The program is `src/example/ex_alert.c`.

```
1  #include <gtk/gtk.h>
2  #include "../tfealert.h"
3
4  static void
5  alert_response_cb (TfeAlert *alert, int response, gpointer user_data) {
6    if (response == TFE_ALERT_RESPONSE_ACCEPT)
7      g_print ("%s\n", tfe_alert_get_button_label (alert));
8    else if (response == TFE_ALERT_RESPONSE_CANCEL)
9      g_print ("Cancel\n");
10   else
11     g_print ("Unexpected␣error\n");
12 }
13
14 static void
15 app_activate (GApplication *application) {
16   GtkWidget *alert;
17   char *title, *message, *btn_label;
18
19   title = "Example␣for␣TfeAlert"; message = "Click␣on␣Cancel␣or␣Accept␣button";
         btn_label = "Accept";
20   alert = tfe_alert_new_with_data (title, message, btn_label);
21   g_signal_connect (TFE_ALERT (alert), "response", G_CALLBACK (alert_response_cb),
         NULL);
22   gtk_window_set_application (GTK_WINDOW (alert), GTK_APPLICATION (application));
23   gtk_window_present (GTK_WINDOW (alert));
24 }
25
26 static void
27 app_startup (GApplication *application) {
28 }
29
30 #define APPLICATION_ID "com.github.ToshioCP.example_tfe_alert"
31
```

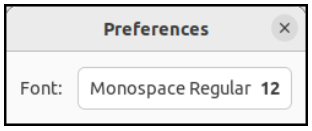


Figure 28: Preference dialog

```
32 int
33 main (int argc, char **argv) {
34   GtkApplication *app;
35   int stat;
36
37   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
38   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
39   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
40   stat =g_application_run (G_APPLICATION (app), argc, argv);
41   g_object_unref (app);
42   return stat;
43 }
```

The “activate” signal handler `app_activate` initializes the alert dialog.

- A TfeAlert instance is created.
- Its “response” signal is connected to the handler `alert_response_cb`.
- TfeAlert class is a sub class of GtkWindow so it can be a top level window that is connected to an application instance. The function `gtk_window_set_application` does that.
- The dialog is shown.

A user clicks on either the cancel button or the accept button. Then, the “response” signal is emitted and the dialog is destroyed. The signal handler `alert_response_cb` checks the response and prints “Accept” or “Cancel”. If an error happens, it prints “Unexpected error”.

You can compile it with meson and ninja.

```
$ cd src/tfe6/example
$ meson setup _build
$ ninja -C _build
$ _build/ex_alert
Accept #<= if you clicked on the accept button
```

21 GtkFontDialogButton and Gsettings

21.1 The preference dialog

If the user clicks on the preference menu, a preference dialog appears.

It has only one button, which is a GtkFontDialogButton widget. You can add more widgets on the dialog but this simple dialog isn’t so bad for the first example program.

If the button is clicked, a FontDialog appears like this.

If the user chooses a font and clicks on the select button, the font is changed.

GtkFontDialogButton and GtkFontDialog are available since GTK version 4.10. They replace GtkFontButton and GtkFontChooserDialog, which are deprecated since 4.10.

21.2 A composite widget

The preference dialog has GtkBox, GtkLabel and GtkFontButton in it and is defined as a composite widget. The following is the template ui file for TfePref.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3   <template class="TfePref" parent="GtkWindow">
```

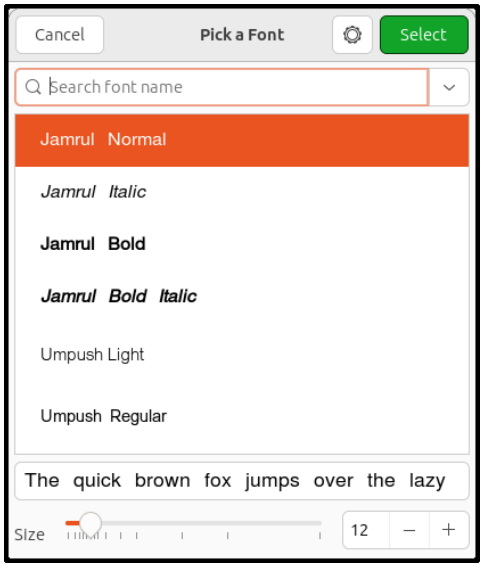


Figure 29: Font dialog

```
4       <property name="title">Preferences</property>
5       <property name="resizable">FALSE</property>
6       <property name="modal">TRUE</property>
7       <child>
8         <object class="GtkBox">
9           <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
10          <property name="spacing">12</property>
11          <property name="halign">GTK_ALIGN_CENTER</property>
12          <property name="margin-start">12</property>
13          <property name="margin-end">12</property>
14          <property name="margin-top">12</property>
15          <property name="margin-bottom">12</property>
16          <child>
17            <object class="GtkLabel">
18              <property name="label">Font:</property>
19              <property name="xalign">1</property>
20            </object>
21          </child>
22          <child>
23            <object class="GtkFontDialogButton" id="font_dialog_btn">
24              <property name="dialog">
25                <object class="GtkFontDialog"/>
26              </property>
27            </object>
28          </child>
29        </object>
30      </child>
31    </template>
32  </interface>
```

- Template tag specifies a composite widget. The class attribute specifies the class name, which is "TfePref". The parent attribute is `GtkWindow`. Therefore. `TfePref` is a child class of `GtkWindow`. A parent attribute is optional but it is recommended to write it explicitly. You can make TfePref as a child of `GtkDialog`, but `GtkDialog` is deprecated since version 4.10.
- There are three properties, title, resizable and modal.
- TfePref has a child widget GtkBox which is horizontal. The box has two children GtkLabel and GtkFontDialogButton.

21.3 The header file

The file `tfepref.h` defines types and declares a public function.

```
1 #pragma once
2
3 #include <gtk/gtk.h>
4
5 #define TFE_TYPE_PREF tfe_pref_get_type ()
6 G_DECLARE_FINAL_TYPE (TfePref, tfe_pref, TFE, PREF, GtkWindow)
7
8 GtkWidget *
9 tfe_pref_new (void);
```

- 5: Defines the type `TFE_TYPE_PREF`, which is a macro replaced by `tfe_pref_get_type ()`.
- 6: The macro `G_DECLAER_FINAL_TYPE` expands to:
  - The function `tfe_pref_get_type ()` is declared.
  - TfePrep type is defined as a typedef of `struct _TfePrep`.
  - TfePrepClass type is defined as a typedef of `struct {GtkWindowClass *parent;}`.
  - Two functions `TFE_PREF ()` and `TFE_IS_PREF ()` is defined.
- 8-9:The function `tfe_pref_new` is declared. It creates a new TfePref instance.

21.4 The C file for composite widget

The following codes are extracted from the file `tfepref.c`.

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

struct _TfePref
{
  GtkWindow parent;
  GtkFontDialogButton *font_dialog_btn;
};

G_DEFINE_FINAL_TYPE (TfePref, tfe_pref, GTK_TYPE_WINDOW);

static void
tfe_pref_dispose (GObject *gobject) {
  TfePref *pref = TFE_PREF (gobject);
  gtk_widget_dispose_template (GTK_WIDGET (pref), TFE_TYPE_PREF);
  G_OBJECT_CLASS (tfe_pref_parent_class)->dispose (gobject);
}

static void
tfe_pref_init (TfePref *pref) {
  gtk_widget_init_template (GTK_WIDGET (pref));
}

static void
tfe_pref_class_init (TfePrefClass *class) {
  G_OBJECT_CLASS (class)->dispose = tfe_pref_dispose;
  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
      "/com/github/ToshioCP/tfe/tfepref.ui");
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfePref,
      font_dialog_btn);
}

GtkWidget *
tfe_pref_new (void) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_PREF, NULL));
}
```

- The structure `_TfePref` has `font_dialog_btn` member. It points the GtkFontDialogButton object

specified in the XML file "tfepref.ui". The member name `font_dialog_btn` must be the same as the GtkFontDialogButton id attribute in the XML file.

- `G_DEFINE_FINAL_TYPE` macro expands to:
  - The declaration of the functions `tfe_pref_init` and `tfe_pref_class_init`. They are defined in the following part of the program.
  - The definition of the variable `tfe_pref_parent_class`.
  - The definition of the function `tfe_pref_get_type`.
- The function `tfe_pref_class_init` initializes the TfePref class. The function `gtk_widget_class_set_template_from_res` initializes the composite widget template from the XML resource. The function `gtk_widget_class_bind_template_child` connects the TfePref structure member `font_dialog_btn` and the GtkFontDialogButton in the XML. The member name and the id attribute value must be the same.
- The function `tfe_pref_init` initializes a newly created instance. The function `gtk_widget_init_template` creates and initializes child widgets.
- The function `tfe_pref_dispose` releases objects. The function `gtk_widget_dispose_template` releases child widgets.

21.5 GtkFontDialogButton and Pango

If the GtkFontDialogButton button is clicked, the GtkFontDialog dialog appears. A user can choose a font on the dialog. If the user clicks on the "select" button, the dialog disappears. And the font information is given to the GtkFontDialogButton instance. The font data is taken with the method `gtk_font_dialog_button_get_font_desc`. It returns a pointer to the PangoFontDescription structure.

Pango is a text layout engine. The documentation is on the internet.

PangoFontDescription is a C structure and it isn't allowed to access directly. The document is here. If you want to retrieve the font information, there are several functions.

- `pango_font_description_to_string` returns a string like "Jamrul Bold Italic Semi-Expanded 12".
- `pango_font_description_get_family` returns a font family like "Jamrul".
- `pango_font_description_get_weight` returns a PangoWeight constant like `PANGO_WEIGHT_BOLD`.
- `pango_font_description_get_style` returns a PangoStyle constant like `PANGO_STYLE_ITALIC`.
- `pango_font_description_get_stretch` returns a PangoStretch constant like `PANGO_STRETCH_SEMI_EXPANDED`.
- `pango_font_description_get_size` returns an integer like `12`. Its unit is point or pixel (device unit). The function `pango_font_description_get_size_is_absolute` returns TRUE if the unit is absolute that means device unit. Otherwise the unit is point.

21.6 GSettings

We want to maintain the font data after the application quits. There are some ways to implement.

- Make a configuration file. For example, a text file "~/.config/tfe/font_desc.cfg" keeps font information.
- Use GSettings object. The basic idea of GSettings are similar to configuration file. Configuration information data is put into a database file.

GSettings is simple and easy to use but a bit hard to understand the concept. This subsection describes the concept first and then how to program it.

21.6.1 GSettings schema

GSettings schema describes a set of keys, value types and some other information. GSettings object uses this schema and it writes/reads the value of a key to/from the right place in the database.

- A schema has an id. The id must be unique. We often use the same string as application id, but schema id and application id are different. You can use different name from application id. Schema id is a string delimited by periods. For example, "com.github.ToshioCP.tfe" is a correct schema id.
- A schema usually has a path. The path is a location in the database. Each key is stored under the path. For example, if a key `font-desc` is defined with a path `/com/github/ToshioCP/tfe/`, the key's location in the database is `/com/github/ToshioCP/tfe/font-desc`. Path is a string begins with and ends with a slash (`/`). And it is delimited by slashes.
- GSettings save information as key-value style. Key is a string begins with a lower case character followed by lower case, digit or dash (`-`) and ends with lower case or digit. No consecutive dashes

are allowed. Values can be any type. GSettings stores values as GVariant type, which can be, for example, integer, double, boolean, string or complex types like an array. The type of values needs to be defined in the schema.
- A default value needs to be set for each key.
- A summery and description can be set for each key optionally.

Schemas are described in an XML format. For example,

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schemalist>
3    <schema path="/com/github/ToshioCP/tfe/" id="com.github.ToshioCP.tfe">
4      <key name="font-desc" type="s">
5        <default>'Monospace 12'</default>
6        <summary>Font</summary>
7        <description>A font to be used for textview.</description>
8      </key>
9    </schema>
10 </schemalist>
```

- 4: The type attribute is “s”. It is GVariant type string. For GVariant type string, see GLib API Reference – GVariant Type Strings. Other common types are:
  - “b”: gboolean
  - “i”: gint32.
  - “d”: double.

Further information is in:

- GLib API Reference – GVariant Format Strings
- GLib API Reference – GVariant Text Format
- GLib API Reference – GVariant
- GLib API Reference – VariantType

21.6.2 Gsettings command

First, let’s try `gsettings` application. It is a configuration tool for GSettings.

```
$ gsettings help
Usage:
  gsettings --version
  gsettings [--schemadir SCHEMADIR] COMMAND [ARGS?]

Commands:
  help                        Show this information
  list-schemas                List installed schemas
  list-relocatable-schemas    List relocatable schemas
  list-keys                   List keys in a schema
  list-children               List children of a schema
  list-recursively            List keys and values, recursively
  range                       Queries the range of a key
  describe                    Queries the description of a key
  get                         Get the value of a key
  set                         Set the value of a key
  reset                       Reset the value of a key
  reset-recursively           Reset all values in a given schema
  writable                    Check if a key is writable
  monitor                     Watch for changes

Use "gsettings help COMMAND" to get detailed help.
```

List schemas.

```
$ gsettings list-schemas
org.gnome.rhythmbox.podcast
ca.desrt.dconf-editor.Demo.Empty
org.gnome.gedit.preferences.ui
```

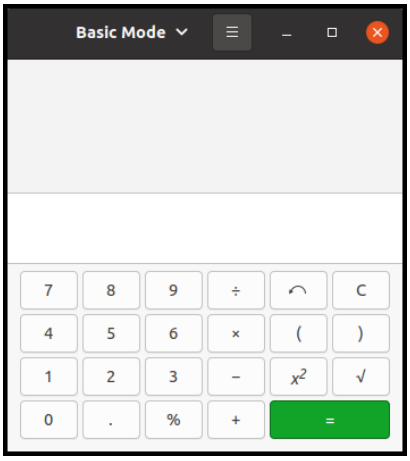


Figure 30: gnome-calculator basic mode

```
org.gnome.evolution-data-server.calendar
org.gnome.rhythmbox.plugins.generic-player

... ...
```

Each line is an id of a schema. Each schema has a key-value configuration data. You can see them with list-recursively command. Let's look at the keys and values of `org.gnome.calculator` schema.

```
$ gsettings list-recursively org.gnome.calculator
org.gnome.calculator accuracy 9
org.gnome.calculator angle-units 'degrees'
org.gnome.calculator base 10
org.gnome.calculator button-mode 'basic'
org.gnome.calculator number-format 'automatic'
org.gnome.calculator precision 2000
org.gnome.calculator refresh-interval 604800
org.gnome.calculator show-thousands false
org.gnome.calculator show-zeroes false
org.gnome.calculator source-currency ''
org.gnome.calculator source-units 'degree'
org.gnome.calculator target-currency ''
org.gnome.calculator target-units 'radian'
org.gnome.calculator window-position (-1, -1)
org.gnome.calculator word-size 64
```

This schema is used by GNOME Calculator. Run the calculator and change the mode, then check the schema again.

```
$ gnome-calculator
```

Change the mode to advanced and quit.

Run gsettings and check the value of `button-mode`.

```
$ gsettings list-recursively org.gnome.calculator

... ...

org.gnome.calculator button-mode 'advanced'

... ...
```

Now we know that GNOME Calculator used gsettings and it has set `button-mode` key to "advanced". The value remains even the calculator quits. So when the calculator runs again, it will appear as an advanced mode.

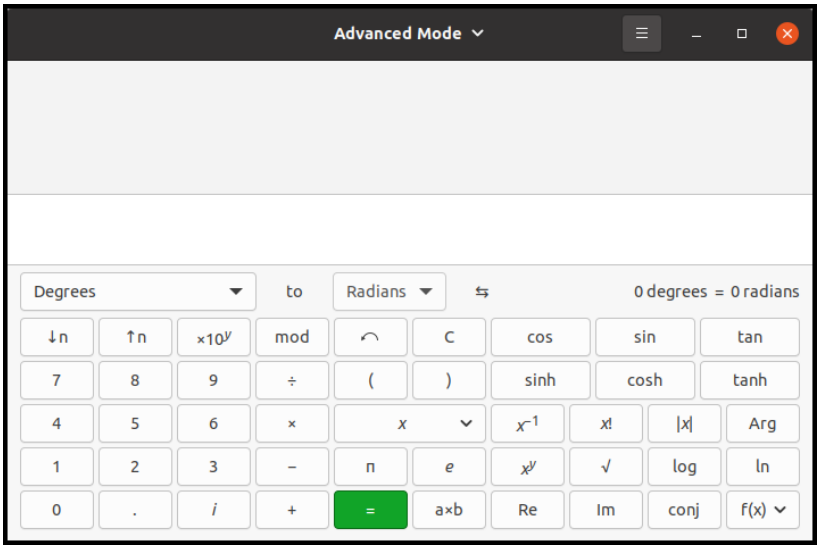


Figure 31: gnome-calculator advanced mode

21.6.3 Glib-compile-schemas utility

GSettings schemas are specified with an XML format. The XML schema files must have the filename extension `.gschema.xml`. The following is the XML schema file for the application `tfe`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schemalist>
3    <schema path="/com/github/ToshioCP/tfe/" id="com.github.ToshioCP.tfe">
4      <key name="font-desc" type="s">
5        <default>'Monospace 12'</default>
6        <summary>Font</summary>
7        <description>A font to be used for textview.</description>
8      </key>
9    </schema>
10 </schemalist>
```

The filename is "com.github.ToshioCP.tfe.gschema.xml". Schema XML filenames are usually the schema id followed by ".gschema.xml" suffix. You can use different name from schema id, but it is not recommended.

- 2: The top level element is `<schemalist>`.
- 3: schema tag has `path` and `id` attributes. A path determines where the settings are stored in the conceptual global tree of settings. An id identifies the schema.
- 4: Key tag has two attributes. Name is the name of the key. Type is the type of the value of the key and it is a GVariant Format String.
- 5: default value of the key `font-desc` is `Monospace 12`.
- 6: Summery and description elements describes the key. They are optional, but it is recommended to add them in the XML file.

The XML file is compiled by glib-compile-schemas. When compiling, `glib-compile-schemas` compiles all the XML files which have ".gschema.xml" file extension in the directory given as an argument. It converts the XML file into a binary file `gschemas.compiled`. Suppose the XML file above is under `tfe6` directory.

```
$ glib-compile-schemas tfe6
```

Then, `gschemas.compiled` is generated under `tfe6`. When you test your application, set `GSETTINGS_SCHEMA_DIR` environment variable so that GSettings objet can find `gschemas.compiled`.

```
$ GSETTINGS_SCHEMA_DIR=(the directory gschemas.compiled is
    located):$GSETTINGS_SCHEMA_DIR (your application name)
```

GSettings object looks for this file by the following process.

- It searches `glib-2.0/schemas` subdirectories of all the directories specified in the environment variable `XDG_DATA_DIRS`. Common directores are `/usr/share/glib-2.0/schemas` and `/usr/local/share/glib-2.0/schemas`.
- If `$HOME/.local/share/glib-2.0/schemas` exists, it is also searched.
- If `GSETTINGS_SCHEMA_DIR` environment variable is defined, it searches all the directories specified in the variable. `GSETTINGS_SCHEMA_DIR` can specify multiple directories delimited by colon (:).

The directories above includes more than one `.gschema.xml` file. Therefore, when you install your application, follow the instruction below to install your schemas.

1. Make `.gschema.xml` file.
2. Copy it to one of the directories above. For example, `$HOME/.local/share/glib-2.0/schemas`.
3. Run `glib-compile-schemas` on the directory. It compiles all the schema files in the directory and creates or updates the database file `gschemas.compiled`.

21.6.4 GSettings object and binding

Now, we go on to the next topic, how to program GSettings.

You need to compile your schema file in advance.

Suppose id, key, class name and a property name are:

- GSettings id: com.github.ToshioCP.sample
- GSettings key: sample_key
- The class name: Sample
- The property to bind: sample_property

The example below uses `g_settings_bind`. If you use it, GSettings key and instance property must have the same the type. In the example, it is assumed that the type of "sample_key" and "sample_property" are the same.

```
GSettings *settings;
Sample *sample_object;

settings = g_settings_new ("com.github.ToshioCP.sample");
sample_object = sample_new ();
g_settings_bind (settings, "sample_key", sample_object, "sample_property",
    G_SETTINGS_BIND_DEFAULT);
```

The function `g_settings_bind` binds the GSettings value and the property of the instance. If the property value is changed, the GSettings value is also changed, and vice versa. The two values are always the same.

The function `g_settings_bind` is simple and easy but it isn't always possible. The type of GSettings are restricted to the type GVariant has. Some property types are out of GVariant. For example, GtkFontDialogButton has "font-desc" property and its type is PangoFontDescription. PangoFontDescription is a C structure and it is wrapped in a boxed type GValue to store in the property. GVariant doesn't support boxed type.

In that case, another function `g_settings_bind_with_mapping` is used. It binds GSettings GVariant value and object property via GValue with mapping functions.

```
void
g_settings_bind_with_mapping (
  GSettings* settings,
  const gchar* key,
  GObject* object,
  const gchar* property,
  GSettingsBindFlags flags, // G_SETTINGS_BIND_DEFAULT is commonly used
  GSettingsBindGetMapping get_mapping, // GSettings => property, See the example
      below
  GSettingsBindSetMapping set_mapping, // property => GSettings, See the example
      below
  gpointer user_data, // NULL if unnecessary
  GDestroyNotify destroy //NULL if unnecessary
)
```

The mapping functions are defined like these:

```
gboolean
(* GSettingsBindGetMapping) (
  GValue* value,
  GVariant* variant,
  gpointer user_data
)

GVariant*
(* GSettingsBindSetMapping) (
  const GValue* value,
  const GVariantType* expected_type,
  gpointer user_data
)
```

The following codes are extracted from `tfepref.c`.

```
1  static gboolean // GSettings => property
2  get_mapping (GValue* value, GVariant* variant, gpointer user_data) {
3    const char *s = g_variant_get_string (variant, NULL);
4    PangoFontDescription *font_desc = pango_font_description_from_string (s);
5    g_value_take_boxed (value, font_desc);
6    return TRUE;
7  }
8
9  static GVariant* // Property => GSettings
10 set_mapping (const GValue* value, const GVariantType* expected_type, gpointer
       user_data) {
11   char*font_desc_string = pango_font_description_to_string (g_value_get_boxed
         (value));
12   return g_variant_new_take_string (font_desc_string);
13 }
14
15 static void
16 tfe_pref_init (TfePref *pref) {
17   gtk_widget_init_template (GTK_WIDGET (pref));
18   pref->settings = g_settings_new ("com.github.ToshioCP.tfe");
19   g_settings_bind_with_mapping (pref->settings, "font-desc", pref->font_dialog_btn,
         "font-desc", G_SETTINGS_BIND_DEFAULT,
20       get_mapping, set_mapping, NULL, NULL);
21 }
```

- 15-21: This function `tfe_pref_init` initializes the new TfePref instance.
- 18: Creates a new GSettings instance. The id is "com.github.ToshioCP.tfe".
- 19-20: Binds the GSettings "font-desc" and the GtkFontDialogButton property "font-desc". The mapping functions are `get_mapping` and `set_mapping`.
- 1-7: The mapping function from GSettings to the property. The first argument `value` is a GValue to be stored in the property. The second argument `variant` is a GVarinat structure that comes from the GSettings value.
- 3: Retrieves a string from the GVariant structure.
- 4: Build a PangoFontDescription structure from the string and assigns its address to `font_desc`.
- 5: Puts `font_desc` into the GValue `value`. The ownership of `font_desc` moves to `value`.
- 6: Returns TRUE that means the mapping succeeds.
- 9-13: The mapping function from the property to GSettings. The first argument `value` holds the property data. The second argument `expected_type` is the type of GVariant that the GSettings value has. It isn't used in this function.
- 11: Gets the PangoFontDescription structure from `value` and converts it to string.
- 12: The string is inserted to a GVariant structure. The ownership of the string `font_desc_string` moves to the returned value.

21.7 C file

The following is the full codes of `tfepref.c`

```c
#include <gtk/gtk.h>
#include "tfepref.h"

struct _TfePref
{
  GtkWindow parent;
  GSettings *settings;
  GtkFontDialogButton *font_dialog_btn;
};

G_DEFINE_FINAL_TYPE (TfePref, tfe_pref, GTK_TYPE_WINDOW);

static void
tfe_pref_dispose (GObject *gobject) {
  TfePref *pref = TFE_PREF (gobject);

  /* GSetting bindings are automatically removed when the object is finalized, so it isn't necessary to unbind them explicitly.*/
  g_clear_object (&pref->settings);
  gtk_widget_dispose_template (GTK_WIDGET (pref), TFE_TYPE_PREF);
  G_OBJECT_CLASS (tfe_pref_parent_class)->dispose (gobject);
}

/* ---------- get_mapping/set_mapping ---------- */
static gboolean // GSettings => property
get_mapping (GValue* value, GVariant* variant, gpointer user_data) {
  const char *s = g_variant_get_string (variant, NULL);
  PangoFontDescription *font_desc = pango_font_description_from_string (s);
  g_value_take_boxed (value, font_desc);
  return TRUE;
}

static GVariant* // Property => GSettings
set_mapping (const GValue* value, const GVariantType* expected_type, gpointer user_data) {
  char*font_desc_string = pango_font_description_to_string (g_value_get_boxed (value));
  return g_variant_new_take_string (font_desc_string);
}

static void
tfe_pref_init (TfePref *pref) {
  gtk_widget_init_template (GTK_WIDGET (pref));
  pref->settings = g_settings_new ("com.github.ToshioCP.tfe");
  g_settings_bind_with_mapping (pref->settings, "font-desc", pref->font_dialog_btn, "font-desc", G_SETTINGS_BIND_DEFAULT,
      get_mapping, set_mapping, NULL, NULL);
}

static void
tfe_pref_class_init (TfePrefClass *class) {
  G_OBJECT_CLASS (class)->dispose = tfe_pref_dispose;
  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class), "/com/github/ToshioCP/tfe/tfepref.ui");
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfePref, font_dialog_btn);
}

GtkWidget *
tfe_pref_new (void) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_PREF, NULL));
}
```

21.8 Test program

There's a test program located at `src/tfe6/test` directory.

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

GSettings *settings;

// "changed::font-desc" signal handler
static void
changed_font_desc_cb (GSettings *settings, char *key, gpointer user_data) {
  char *s;
  s = g_settings_get_string (settings, key);
  g_print ("%s\n", s);
  g_free (s);
}

static void
app_shutdown (GApplication *application) {
  g_object_unref (settings);
}

static void
app_activate (GApplication *application) {
  GtkWidget *pref = tfe_pref_new ();

  gtk_window_set_application (GTK_WINDOW (pref), GTK_APPLICATION (application));
  gtk_window_present (GTK_WINDOW (pref));
}

static void
app_startup (GApplication *application) {
  settings = g_settings_new ("com.github.ToshioCP.tfe");
  g_signal_connect (settings, "changed::font-desc", G_CALLBACK
      (changed_font_desc_cb), NULL);
  g_print ("%s\n", "Change␣the␣font␣with␣the␣font␣button.␣Then␣the␣new␣font␣will␣be␣
      printed␣out.\n");
}

#define APPLICATION_ID "com.github.ToshioCP.test_tfe_pref"

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

  app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
  g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
  g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
  g_signal_connect (app, "shutdown", G_CALLBACK (app_shutdown), NULL);
  stat =g_application_run (G_APPLICATION (app), argc, argv);
  g_object_unref (app);
  return stat;
}
```

This program sets its active window to TfePref instance, which is a child object of GtkWindow.

It sets the "changed::font-desc" signal handler in the startup function. The process from the user's font selection to the handler is:

- The user clicked on the GtkFontDialogButton and GtkFontDialog appears.
- He/she selects a new font.
- The "font-desc" property of the GtkFontDialogButton instance is changed.
- The value of "font-desc" key on the GSettings database is changed since it is bound to the property.

- The “changed::font-desc” signal on the GSettings instance is emitted.
- The handler is called.

The program building is divided into four steps.

- Compile the schema file
- Compile the XML file to a resource (C source file)
- Compile the C files
- Run the executable file

Commands are shown in the next four sub-subsections. You don’t need to try them. The final sub-subsection shows the meson-ninja way, which is the easiest.

21.8.1 Compile the schema file

```
$ cd src/tef6/test
$ cp ../com.github.ToshioCP.tfe.gschema.xml com.github.ToshioCP.tfe.gschema.xml
$ glib-compile-schemas .
```

Be careful. The commands `glib-compile-schemas` has an argument “.”, which means the current directory. This results in creating `gschemas.compiled` file.

21.8.2 Compile the XML file

```
$ glib-compile-resources --sourcedir=.. --generate-source --target=resource.c
    ../tfe.gresource.xml
```

21.8.3 Compile the C file

```
$ gcc `pkg-config --cflags gtk4` test_pref.c ../tfepref.c resource.c `pkg-config
    --libs gtk4`
```

21.8.4 Run the executable file

```
$ GSETTINGS_SCHEMA_DIR=. ./a.out

Jamrul Italic Semi-Expanded 12 # <= select Jamrul Italic 12
Monospace 12 #<= select Monospace Regular 12
```

21.8.5 Meson-ninja way

Meson wraps up the commands above. Create the following text and save it to `meson.build`.

Note: Gtk4-tutorial repository has meson.build file that defines several tests. So you can try it instead of the following text.

```
project('tfe_pref_test', 'c')

gtkdep = dependency('gtk4')

gnome=import('gnome')
resources = gnome.compile_resources('resources','../tfe.gresource.xml', source_dir:
    '..')
gnome.compile_schemas(build_by_default: true, depend_files:
    'com.github.ToshioCP.tfe.gschema.xml')

executable('test_pref', ['test_pref.c', '../tfepref.c'], resources, dependencies:
    gtkdep, export_dynamic: true, install: false)
```

- Project name is ‘tfe_pref_test’ and it is written in C language.
- It depends on GTK4 library.
- It uses GNOME module. Modules are prepared by Meson.
- GNOME module has `compile_resources` method. When you call this method, you need the prefix “gnome.”.

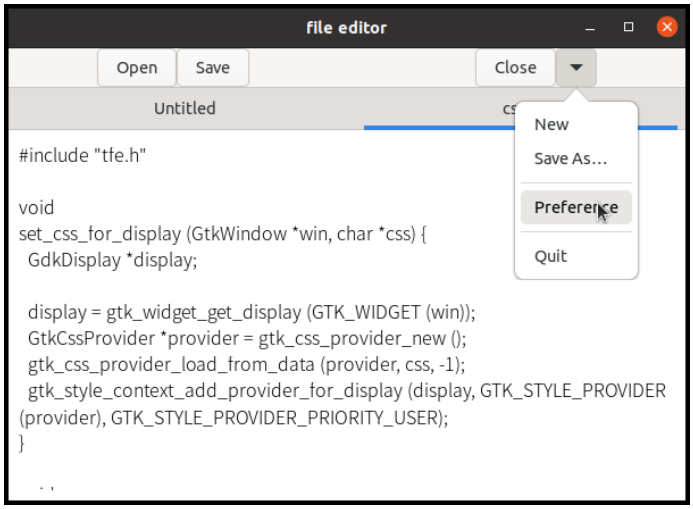


Figure 32: tfe6

  - The target filename is resources.
  - The definition XML file is '../tfe.gresource.xml'.
  - The source dir is '.'. All the ui files are located there.
- GNOME module has `compile_schemas` method. It compiles the schema file 'com.github.ToshioCP.tfe.gschema.xml'. You need to copy '../com.github.ToshioCP.tfe.gschema.xml' to the current directory in advance.
- It creates an executable file 'test_pref'. The source files are 'test_pref.c', '../tfepref.c' and `resources`, which is made by `gnome.compile_resources`. It depends on `gtkdep`, which is GTK4 library. The symbols are exported and no installation support.

Type like this to build and test the program.

```
$ cd src/tef6/test
$ cp ../com.github.ToshioCP.tfe.gschema.xml com.github.ToshioCP.tfe.gschema.xml
$ meson setup _build
$ ninja -C _build
$ GSETTINGS_SCHEMA_DIR=_build _build/test_pref
```

A window appears and you can choose a font via GtkFontDialog. If you select a new font, the font string is output through the standard output.

22 TfeWindow

22.1 The Tfe window and XML files

The following is the window of Tfe.

- Open, save and close buttons are placed on the toolbar. In addition, GtkMenuButton is added to the toolbar. This button shows a popup menu when clicked on. Here, popup means widely, including pull-down menu.
- New, save-as, preference and quit items are put into the menu.

This makes the most frequently used operation bound to the tool bar buttons. And the others are stored in behind the menus. So, it is more practical.

The window is a composite widget. The definition is described in the XML file `tfewindow.ui`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <template class="TfeWindow" parent="GtkApplicationWindow">
4      <property name="title">Text File Editor</property>
5      <property name="default-width">600</property>
6      <property name="default-height">400</property>
```

```
7       <child>
8         <object class="GtkBox" id="boxv">
9           <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
10          <child>
11            <object class="GtkBox" id="boxh">
12              <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
13              <child>
14                <object class="GtkLabel">
15                  <property name="width-chars">10</property>
16                </object>
17              </child>
18              <child>
19                <object class="GtkButton">
20                  <property name="label">Open</property>
21                  <property name="action-name">win.open</property>
22                </object>
23              </child>
24              <child>
25                <object class="GtkButton">
26                  <property name="label">Save</property>
27                  <property name="action-name">win.save</property>
28                </object>
29              </child>
30              <child>
31                <object class="GtkLabel">
32                  <property name="hexpand">TRUE</property>
33                </object>
34              </child>
35              <child>
36                <object class="GtkButton">
37                  <property name="label">Close</property>
38                  <property name="action-name">win.close</property>
39                </object>
40              </child>
41              <child>
42                <object class="GtkMenuButton" id="btnm">
43                  <property name="direction">down</property>
44                  <property name="icon-name">open-menu-symbolic</property>
45                </object>
46              </child>
47              <child>
48                <object class="GtkLabel">
49                  <property name="width-chars">10</property>
50                </object>
51              </child>
52            </object>
53          </child>
54          <child>
55            <object class="GtkNotebook" id="nb">
56              <property name="scrollable">TRUE</property>
57              <property name="hexpand">TRUE</property>
58              <property name="vexpand">TRUE</property>
59            </object>
60          </child>
61        </object>
62      </child>
63    </template>
64  </interface>
```

- Three buttons “Open”, “Save” and “Close” are defined. You can use two ways to catch the button click event. The one is “clicked” signal and the other is to register an action to the button. The first way is simple. You can connects the signal and your handler directly. The second way is like menu items. When the button is clicked, the corresponding action is activated. It is a bit complicated because you need to create an action and its “activate” handler in advance. But one advantage is you

can connect two or more things to the action. For example, an accelerator can be connected to the action. Accelerators are keys that connects to actions. For example, Ctrl+O is often connected to a file open action. So, both open button and Ctrl+O activates an open action. The latter way is used in the XML file above.

- You can specify a theme icon to GtkMenuButton with "icon-name" poperty. The "open-menu-symbolic" is an image that is called hamburger menu.

The `menu.ui` XML file defines the menu for GtkMenuButton.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <menu id="menu">
4      <section>
5        <item>
6          <attribute name="label">New</attribute>
7          <attribute name="action">win.new</attribute>
8        </item>
9        <item>
10         <attribute name="label">Save …As</attribute>
11         <attribute name="action">win.saveas</attribute>
12       </item>
13     </section>
14     <section>
15       <item>
16         <attribute name="label">Preference</attribute>
17         <attribute name="action">win.pref</attribute>
18       </item>
19     </section>
20     <section>
21       <item>
22         <attribute name="label">Quit</attribute>
23         <attribute name="action">win.close-all</attribute>
24       </item>
25     </section>
26   </menu>
27 </interface>
```

There are four menu items and they are connected to actions.

22.2 The header file

The following is the codes of `tfewindow.h`.

```
1  #pragma once
2
3  #include <gtk/gtk.h>
4
5  #define TFE_TYPE_WINDOW tfe_window_get_type ()
6  G_DECLARE_FINAL_TYPE (TfeWindow, tfe_window, TFE, WINDOW, GtkApplicationWindow)
7
8  void
9  tfe_window_notebook_page_new (TfeWindow *win);
10
11 void
12 tfe_window_notebook_page_new_with_files (TfeWindow *win, GFile **files, int n_files);
13
14 GtkWidget *
15 tfe_window_new (GtkApplication *app);
```

- 5-6: `TFE_TYPE_WINDOW` definition and the `G_DECLARE_FINAL_TYPE` macro.
- 8-15: Public functions. The first two functions creates a notebook page and the last function creates a window.

22.3 C file

22.3.1 A composite widget

The following codes are extracted from `tfewindow.c`.

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

struct _TfeWindow {
  GtkApplicationWindow parent;
  GtkMenuButton *btnm;
  GtkNotebook *nb;
  gboolean is_quit;
};

G_DEFINE_FINAL_TYPE (TfeWindow, tfe_window, GTK_TYPE_APPLICATION_WINDOW);

static void
tfe_window_dispose (GObject *gobject) {
  gtk_widget_dispose_template (GTK_WIDGET (gobject), TFE_TYPE_WINDOW);
  G_OBJECT_CLASS (tfe_window_parent_class)->dispose (gobject);
}

static void
tfe_window_init (TfeWindow *win) {
  GtkBuilder *build;
  GMenuModel *menu;

  gtk_widget_init_template (GTK_WIDGET (win));

  build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfe/menu.ui");
  menu = G_MENU_MODEL (gtk_builder_get_object (build, "menu"));
  gtk_menu_button_set_menu_model (win->btnm, menu);
  g_object_unref(build);
... ... ...
}

static void
tfe_window_class_init (TfeWindowClass *class) {
  GObjectClass *object_class = G_OBJECT_CLASS (class);

  object_class->dispose = tfe_window_dispose;
  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
      "/com/github/ToshioCP/tfe/tfewindow.ui");
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeWindow, btnm);
  gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeWindow, nb);
}

GtkWidget *
tfe_window_new (GtkApplication *app) {
  return GTK_WIDGET (g_object_new (TFE_TYPE_WINDOW, "application", app, NULL));
}
```

The program above is similar to `tfealert.c` and `tfepref.c`. It uses the same way to build a composite widget. But there's one thing new. It is menu. The menu is built from the XML resource `menu.ui` and inserted into the menu button. It is done in the instance initialization function `tfe_window_init`.

22.3.2 Actions

Actions can belong to an application or window. Tfe only has one top window and all the actions are registered in the window. For example, "close-all" action destroys the top level window and that brings the application to quit. You can make "app.quit" action instead of "win.close-all". It's your choice. If your application has two or more windows, both "app.quit" and "win:close-all", which closes all the notebook

pages on the window, may be necessary. Anyway, you need to consider that each action should belong to the application or a window.

Actions are defined in the instance initialization function.

```
static void
tfe_window_init (TfeWindow *win) {
... ... ...
/* ----- action ----- */
  const GActionEntry win_entries[] = {
    { "open", open_activated, NULL, NULL, NULL },
    { "save", save_activated, NULL, NULL, NULL },
    { "close", close_activated, NULL, NULL, NULL },
    { "new", new_activated, NULL, NULL, NULL },
    { "saveas", saveas_activated, NULL, NULL, NULL },
    { "pref", pref_activated, NULL, NULL, NULL },
    { "close-all", close_all_activated, NULL, NULL, NULL }
  };
  g_action_map_add_action_entries (G_ACTION_MAP (win), win_entries, G_N_ELEMENTS
      (win_entries), win);
... ... ...
}
```

Two things are necessary, an array and the `g_action_map_add_action_entries` function.

- The element of the array is the GActionEntry structure. The structure has the following members:
  - an action name
  - a handler for the activate signal
  - the type of the parameter or NULL for no parameter
  - the initial state for the action
  - a handler for the change-state signal
- The actions above are stateless and have no parameters. So, the third parameter and after are all NULL.
- The function `g_action_map_add_action_entries` adds the actions in the `win_entries` array to the action map `win`. The last argument `win` is the user_data, which is the last argument of handlers.
- All the handlers are in `tfewindow.c` program and shown in the following subsections.

22.3.3 The handlers of the actions

open_activated The callback function `open_activated` is an activate signal handler on "open" action.

```
1  static void
2  open_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
3    TfeWindow *win = TFE_WINDOW (user_data);
4    GtkWidget *tv = tfe_text_view_new ();
5
6    g_signal_connect (TFE_TEXT_VIEW (tv), "open-response", G_CALLBACK
         (open_response_cb), win);
7    tfe_text_view_open (TFE_TEXT_VIEW (tv), GTK_WINDOW (win));
8  }
```

It connects the "open-response" signal on the newly created TfeTextView instance and just calls `tfe_text_view_open`. It leaves the rest of the task to the signal handler `open_response_cb`.

```
1  static void
2  open_response_cb (TfeTextView *tv, int response, gpointer user_data) {
3    TfeWindow *win = TFE_WINDOW (user_data);
4    GFile *file;
5    char *filename;
6
7    if (response != TFE_OPEN_RESPONSE_SUCCESS) {
8      g_object_ref_sink (tv);
9      g_object_unref (tv);
10   }else if (! G_IS_FILE (file = tfe_text_view_get_file (tv))) {
11     g_object_ref_sink (tv);
```

```
    g_object_unref (tv);
  }else {
    filename = g_file_get_basename (file);
    g_object_unref (file);
    notebook_page_build (win, GTK_WIDGET (tv), filename);
    g_free (filename);
  }
}
```

If the TfeTextView instance failed to read a file, it destroys the instance with `g_object_ref_sink` and `g_object_unref`. Since newly created widgets are floating, you need to convert the floating reference to the normal reference before you release it. The conversion is done with `g_object_ref_sink`.

If the instance successfully read the file, it calls `notebook_page_build` to build a notebook page and add it to the GtkNotebook object.

```
static void
notebook_page_build (TfeWindow *win, GtkWidget *tv, char *filename) {
  // The arguments win, tb and filename are owned by the caller.
  // If tv has a floating reference, it is consumed by the function.
  GtkWidget *scr = gtk_scrolled_window_new ();
  GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
  GtkNotebookPage *nbp;
  GtkWidget *lab;
  int i;

  gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
  gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
  lab = gtk_label_new (filename);
  i = gtk_notebook_append_page (win->nb, scr, lab);
  nbp = gtk_notebook_get_page (win->nb, scr);
  g_object_set (nbp, "tab-expand", TRUE, NULL);
  gtk_notebook_set_current_page (win->nb, i);
  g_signal_connect (GTK_TEXT_VIEW (tv), "change-file", G_CALLBACK (file_changed_cb),
      win->nb);
  g_signal_connect (tb, "modified-changed", G_CALLBACK (modified_changed_cb), tv);
}
```

This function is a kind of library function and it is called from the different three places.

This function creates a new GtkScrolledWindow instance and sets its child to `tv`. Then it appends it to the GtkNotebook instance `win->nb`. And it sets the tab label to the filename.

After the building, it connects two signals and handlers.

- "change-file" signal and `file_changed_cb` handler. If the TfeTextView instance changes the file, the handler is called and the notebook page tab is updated.
- "modified-changed" signal and `modified_changed_cb` handler. If the text in the buffer of TfeTextView instance is modified, an asterisk is added at the beginning of the filename of the notebook page tab. If the text is saved to the file, the asterisk is removed. The asterisk tells the user that the text has been modified or not.

```
static void
file_changed_cb (TfeTextView *tv, gpointer user_data) {
  GtkNotebook *nb =  GTK_NOTEBOOK (user_data);
  GtkWidget *scr;
  GtkWidget *label;
  GFile *file;
  char *filename;

  file = tfe_text_view_get_file (tv);
  scr = gtk_widget_get_parent (GTK_WIDGET (tv));
  if (G_IS_FILE (file)) {
    filename = g_file_get_basename (file);
    g_object_unref (file);
```

```
  } else
    filename = get_untitled ();
  label = gtk_label_new (filename);
  g_free (filename);
  gtk_notebook_set_tab_label (GTK_NOTEBOOK (nb), scr, label);
}

static void
modified_changed_cb (GtkTextBuffer *tb, gpointer user_data) {
  TfeTextView *tv = TFE_TEXT_VIEW (user_data);
  GtkWidget *scr = gtk_widget_get_parent (GTK_WIDGET (tv));
  GtkWidget *nb =  gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_NOTEBOOK);
  GtkWidget *label;
  GFile *file;
  char *filename;
  char *text;

  file = tfe_text_view_get_file (tv);
  filename = g_file_get_basename (file);
  if (gtk_text_buffer_get_modified (tb))
    text = g_strdup_printf ("*%s", filename);
  else
    text = g_strdup (filename);
  g_object_unref (file);
  g_free (filename);
  label = gtk_label_new (text);
  g_free (text);
  gtk_notebook_set_tab_label (GTK_NOTEBOOK (nb), scr, label);
}
```

save_activated The callback function `save_activated` is an activate signal handler on "save" action.

```
static void
save_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
  TfeWindow *win = TFE_WINDOW (user_data);
  TfeTextView *tv = get_current_textview (win->nb);

  tfe_text_view_save (TFE_TEXT_VIEW (tv));
}
```

This function gets the current TfeTextView instance with the function `get_current_textview`. And it just calls `tfe_text_view_save`.

```
static TfeTextView *
get_current_textview (GtkNotebook *nb) {
  int i;
  GtkWidget *scr;
  GtkWidget *tv;

  i = gtk_notebook_get_current_page (nb);
  scr = gtk_notebook_get_nth_page (nb, i);
  tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
  return TFE_TEXT_VIEW (tv);
}
```

close_activated The callback function `close_activated` is an activate signal handler on "close" action. It closes the current notebook page.

```
static void
close_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
  TfeWindow *win = TFE_WINDOW (user_data);
  TfeTextView *tv;
  GtkTextBuffer *tb;
  GtkWidget *alert;
```

```
7
8     tv = get_current_textview (win->nb);
9     tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
10    if (! gtk_text_buffer_get_modified (tb)) /* is saved? */
11      notebook_page_close (win);
12    else {
13      win->is_quit = FALSE;
14      alert = tfe_alert_new_with_data ("Are you sure?", "Contents aren't saved
            yet.\nAre you sure to close?", "Close");
15      gtk_window_set_transient_for (GTK_WINDOW (alert), GTK_WINDOW (win));
16      g_signal_connect (TFE_ALERT (alert), "response", G_CALLBACK (alert_response_cb),
            win);
17      gtk_window_present (GTK_WINDOW (alert));
18    }
19  }
```

If the text in the current page has been saved, it calls `notebook_page_close` to close the page. Otherwise, it sets `win->is_quit` to FALSE and show the alert dialog. The "response" signal on the dialog is connected to the handler `alert_response_cb`.

```
1   static void
2   notebook_page_close (TfeWindow *win){
3     int i;
4
5     if (gtk_notebook_get_n_pages (win->nb) == 1)
6       gtk_window_destroy (GTK_WINDOW (win));
7     else {
8       i = gtk_notebook_get_current_page (win->nb);
9       gtk_notebook_remove_page (win->nb, i);
10    }
11  }
```

If the notebook has only one page, it destroys the window and the application quits. Otherwise, it removes the current page.

```
1   static void
2   alert_response_cb (TfeAlert *alert, int response_id, gpointer user_data) {
3     TfeWindow *win = TFE_WINDOW (user_data);
4
5     if (response_id == TFE_ALERT_RESPONSE_ACCEPT) {
6       if (win->is_quit)
7         gtk_window_destroy(GTK_WINDOW (win));
8       else
9         notebook_page_close (win);
10    }
11  }
```

If the user clicked on the cacel button, it does nothing. If the user clicked on the accept button, which is the same as close button, it calls `notebook_page_close`. Note that `win->is_quit` has been set to FALSE in the `close_activated` function.

new_activated The callback function `new_activated` is an activate signal handler on "new" action.

```
1   static void
2   new_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
3     TfeWindow *win = TFE_WINDOW (user_data);
4
5     tfe_window_notebook_page_new (win);
6   }
```

It just calls `tfe_window_notebook_page_new`, which is a public method of TfeWindow.

```
1   void
2   tfe_window_notebook_page_new (TfeWindow *win) {
3     GtkWidget *tv;
```

```
4     char *filename;
5
6     tv = tfe_text_view_new ();
7     filename = get_untitled ();
8     notebook_page_build (win, tv, filename);
9     g_free (filename);
10  }
```

This function creates a new TfeTextView instance, “Untitled” family string and calls `notebook_page_build`.

saveas_activated The callback function `saveas_activated` is an activate signal handler on “saveas” action.

```
1  static void
2  saveas_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
3    TfeWindow *win = TFE_WINDOW (user_data);
4    TfeTextView *tv = get_current_textview (win->nb);
5
6    tfe_text_view_saveas (TFE_TEXT_VIEW (tv));
7  }
```

This function gets the current page TfeTextView instance and calls `tfe_text_view_saveas`.

pref_activated The callback function `pref_activated` is an activate signal handler on “pref” action.

```
1  static void
2  pref_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
3    TfeWindow *win = TFE_WINDOW (user_data);
4    GtkWidget *pref;
5
6    pref = tfe_pref_new ();
7    gtk_window_set_transient_for (GTK_WINDOW (pref), GTK_WINDOW (win));
8    gtk_window_present (GTK_WINDOW (pref));
9  }
```

This function creates a TfePref instance, which is a dialog, and sets the transient parent window to `win`. And it shows the dialog.

close_all_activated The callback function `close_all_activated` is an activate signal handler on “close_all” action.

```
1  static void
2  close_all_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data)
       {
3    TfeWindow *win = TFE_WINDOW (user_data);
4
5    if (close_request_cb (win) == FALSE)
6      gtk_window_destroy (GTK_WINDOW (win));
7  }
```

It first calls the function `close_request_cb`. It is a callback function for the “close-request” signal on the top window. It returns FALSE if all the texts have been saved. Otherwise it returns TRUE.

Therefore, function `close_all_activated` destroys the top window if all the texts have been saved. Otherwise it does nothing. But, the function `close_request_cb` shows an alert dialog and if the user clicks on the accept button, the window will be destroyed.

22.3.4 Window “close-request” signal

GtkWindow has a “close-request” signal and it is emitted when the close button, which is x-shaped button at the top right corner, is clicked on. And the user handler is called before the default handler. If the user handler returns TRUE, the rest of the close process is skipped. If it returns FALSE, the rest will go on and the window will be destroyed.

```
static gboolean
close_request_cb (TfeWindow *win) {
  TfeAlert *alert;

  if (is_saved_all (win->nb))
    return FALSE;
  else {
    win->is_quit = TRUE;
    alert = TFE_ALERT (tfe_alert_new_with_data ("Are you sure?", "Contents aren't
        saved yet.\nAre you sure to quit?", "Quit"));
    gtk_window_set_transient_for (GTK_WINDOW (alert), GTK_WINDOW (win));
    g_signal_connect (TFE_ALERT (alert), "response", G_CALLBACK (alert_response_cb),
        win);
    gtk_window_present (GTK_WINDOW (alert));
    return TRUE;
  }
}
```

First, it calls `is_saved_all` and checks if the texts have been saved. If so, it returns FALSE and the close process continues. Otherwise, it sets `win->is_quit` to TRUE and shows an alert dialog. When the user clicks on the accept or cancel button, the dialog disappears and "response" signal is emitted. Then, the handler `alert_response_cb` is called. It destroys the top window if the user clicked on the accept button since `win->is_quit` is TRUE. Otherwise it does nothing.

```
static gboolean
is_saved_all (GtkNotebook *nb) {
  int i, n;
  GtkWidget *scr;
  GtkWidget *tv;
  GtkTextBuffer *tb;

  n = gtk_notebook_get_n_pages (nb);
  for (i = 0; i < n; ++i) {
    scr = gtk_notebook_get_nth_page (nb, i);
    tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
    tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
    if (gtk_text_buffer_get_modified (tb))
      return FALSE;
  }
  return TRUE;
}
```

22.3.5 Public functions

There are three public functions.

- `void tfe_window_notebook_page_new (TfeWindow *win)`
- `void tfe_window_notebook_page_new_with_files (TfeWindow *win, GFile **files, int n_files)`
- `GtkWidget *tfe_window_new (GtkApplication *app)`

The first function is called when the application emits the "activate" signal. The second is for "open" signal. It is given three arguments and they are owned by the caller.

```
void
tfe_window_notebook_page_new_with_files (TfeWindow *win, GFile **files, int n_files)
    {
  int i;
  GtkWidget *tv;
  char *filename;

  for (i = 0; i < n_files; i++)
    if ((tv = tfe_text_view_new_with_file (*(files+i))) != NULL) {
      filename = g_file_get_basename (*(files+i));
      notebook_page_build (win, tv, filename);
      g_free (filename);
```

```
12      }
13    if (gtk_notebook_get_n_pages (win->nb) == 0)
14      tfe_window_notebook_page_new (win);
15  }
```

This function has a loop for the array `files`. It creates TfeTextView instance with the text from each file. And build a page with it.

If an error happens and no page is created, it creates a new empty page.

22.3.6 Full codes of tfewindow.c

The following is the full source codes of `tfewindow.c`.

```
1   #include <gtk/gtk.h>
2   #include "tfewindow.h"
3   #include "tfepref.h"
4   #include "tfealert.h"
5   #include "../tfetextview/tfetextview.h"
6
7   struct _TfeWindow {
8     GtkApplicationWindow parent;
9     GtkMenuButton *btnm;
10    GtkNotebook *nb;
11    gboolean is_quit;
12  };
13
14  G_DEFINE_FINAL_TYPE (TfeWindow, tfe_window, GTK_TYPE_APPLICATION_WINDOW);
15
16  /* Low level functions */
17
18  /* Create a new untitled string */
19  /* The returned string should be freed with g_free() when no longer needed. */
20  static char*
21  get_untitled () {
22    static int c = -1;
23    if (++c == 0)
24      return g_strdup_printf("Untitled");
25    else
26      return g_strdup_printf ("Untitled%u", c);
27  }
28
29  /* The returned object is owned by the scrolled window. */
30  /* The caller won't get the ownership and mustn't release it. */
31  static TfeTextView *
32  get_current_textview (GtkNotebook *nb) {
33    int i;
34    GtkWidget *scr;
35    GtkWidget *tv;
36
37    i = gtk_notebook_get_current_page (nb);
38    scr = gtk_notebook_get_nth_page (nb, i);
39    tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
40    return TFE_TEXT_VIEW (tv);
41  }
42
43  /* This call back is called when a TfeTextView instance emits a "change-file"
        signal. */
44  static void
45  file_changed_cb (TfeTextView *tv, gpointer user_data) {
46    GtkNotebook *nb =  GTK_NOTEBOOK (user_data);
47    GtkWidget *scr;
48    GtkWidget *label;
49    GFile *file;
50    char *filename;
```

```
51
52    file = tfe_text_view_get_file (tv);
53    scr = gtk_widget_get_parent (GTK_WIDGET (tv));
54    if (G_IS_FILE (file)) {
55      filename = g_file_get_basename (file);
56      g_object_unref (file);
57    } else
58      filename = get_untitled ();
59    label = gtk_label_new (filename);
60    g_free (filename);
61    gtk_notebook_set_tab_label (GTK_NOTEBOOK (nb), scr, label);
62  }
63
64  static void
65  modified_changed_cb (GtkTextBuffer *tb, gpointer user_data) {
66    TfeTextView *tv = TFE_TEXT_VIEW (user_data);
67    GtkWidget *scr = gtk_widget_get_parent (GTK_WIDGET (tv));
68    GtkWidget *nb =  gtk_widget_get_ancestor (GTK_WIDGET (tv), GTK_TYPE_NOTEBOOK);
69    GtkWidget *label;
70    GFile *file;
71    char *filename;
72    char *text;
73
74    file = tfe_text_view_get_file (tv);
75    filename = g_file_get_basename (file);
76    if (gtk_text_buffer_get_modified (tb))
77      text = g_strdup_printf ("*%s", filename);
78    else
79      text = g_strdup (filename);
80    g_object_unref (file);
81    g_free (filename);
82    label = gtk_label_new (text);
83    g_free (text);
84    gtk_notebook_set_tab_label (GTK_NOTEBOOK (nb), scr, label);
85  }
86
87  static gboolean
88  is_saved_all (GtkNotebook *nb) {
89    int i, n;
90    GtkWidget *scr;
91    GtkWidget *tv;
92    GtkTextBuffer *tb;
93
94    n = gtk_notebook_get_n_pages (nb);
95    for (i = 0; i < n; ++i) {
96      scr = gtk_notebook_get_nth_page (nb, i);
97      tv = gtk_scrolled_window_get_child (GTK_SCROLLED_WINDOW (scr));
98      tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
99      if (gtk_text_buffer_get_modified (tb))
100       return FALSE;
101   }
102   return TRUE;
103 }
104
105 static void
106 notebook_page_close (TfeWindow *win){
107   int i;
108
109   if (gtk_notebook_get_n_pages (win->nb) == 1)
110     gtk_window_destroy (GTK_WINDOW (win));
111   else {
112     i = gtk_notebook_get_current_page (win->nb);
113     gtk_notebook_remove_page (win->nb, i);
114   }
```

```
115 }
116
117 static void
118 notebook_page_build (TfeWindow *win, GtkWidget *tv, char *filename) {
119   // The arguments win, tb and filename are owned by the caller.
120   // If tv has a floating reference, it is consumed by the function.
121   GtkWidget *scr = gtk_scrolled_window_new ();
122   GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
123   GtkNotebookPage *nbp;
124   GtkWidget *lab;
125   int i;
126
127   gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (tv), GTK_WRAP_WORD_CHAR);
128   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), tv);
129   lab = gtk_label_new (filename);
130   i = gtk_notebook_append_page (win->nb, scr, lab);
131   nbp = gtk_notebook_get_page (win->nb, scr);
132   g_object_set (nbp, "tab-expand", TRUE, NULL);
133   gtk_notebook_set_current_page (win->nb, i);
134   g_signal_connect (GTK_TEXT_VIEW (tv), "change-file", G_CALLBACK (file_changed_cb),
        win->nb);
135   g_signal_connect (tb, "modified-changed", G_CALLBACK (modified_changed_cb), tv);
136 }
137
138 static void
139 open_response_cb (TfeTextView *tv, int response, gpointer user_data) {
140   TfeWindow *win = TFE_WINDOW (user_data);
141   GFile *file;
142   char *filename;
143
144   if (response != TFE_OPEN_RESPONSE_SUCCESS) {
145     g_object_ref_sink (tv);
146     g_object_unref (tv);
147   }else if (! G_IS_FILE (file = tfe_text_view_get_file (tv))) {
148     g_object_ref_sink (tv);
149     g_object_unref (tv);
150   }else {
151     filename = g_file_get_basename (file);
152     g_object_unref (file);
153     notebook_page_build (win, GTK_WIDGET (tv), filename);
154     g_free (filename);
155   }
156 }
157
158 /* alert response signal handler */
159 static void
160 alert_response_cb (TfeAlert *alert, int response_id, gpointer user_data) {
161   TfeWindow *win = TFE_WINDOW (user_data);
162
163   if (response_id == TFE_ALERT_RESPONSE_ACCEPT) {
164     if (win->is_quit)
165       gtk_window_destroy(GTK_WINDOW (win));
166     else
167       notebook_page_close (win);
168   }
169 }
170
171 /* ----- Close request on the top window ----- */
172 /* ----- The signal is emitted when the close button is clicked. ----- */
173 static gboolean
174 close_request_cb (TfeWindow *win) {
175   TfeAlert *alert;
176
177   if (is_saved_all (win->nb))
```

```
178       return FALSE;
179     else {
180       win->is_quit = TRUE;
181       alert = TFE_ALERT (tfe_alert_new_with_data ("Are␣you␣sure?", "Contents␣aren't␣
              saved␣yet.\nAre␣you␣sure␣to␣quit?", "Quit"));
182       gtk_window_set_transient_for (GTK_WINDOW (alert), GTK_WINDOW (win));
183       g_signal_connect (TFE_ALERT (alert), "response", G_CALLBACK (alert_response_cb),
              win);
184       gtk_window_present (GTK_WINDOW (alert));
185       return TRUE;
186     }
187   }
188
189   /* ----- action activated handlers ----- */
190   static void
191   open_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
192     TfeWindow *win = TFE_WINDOW (user_data);
193     GtkWidget *tv = tfe_text_view_new ();
194
195     g_signal_connect (TFE_TEXT_VIEW (tv), "open-response", G_CALLBACK
            (open_response_cb), win);
196     tfe_text_view_open (TFE_TEXT_VIEW (tv), GTK_WINDOW (win));
197   }
198
199   static void
200   save_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
201     TfeWindow *win = TFE_WINDOW (user_data);
202     TfeTextView *tv = get_current_textview (win->nb);
203
204     tfe_text_view_save (TFE_TEXT_VIEW (tv));
205   }
206
207   static void
208   close_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
209     TfeWindow *win = TFE_WINDOW (user_data);
210     TfeTextView *tv;
211     GtkTextBuffer *tb;
212     GtkWidget *alert;
213
214     tv = get_current_textview (win->nb);
215     tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
216     if (! gtk_text_buffer_get_modified (tb)) /* is saved? */
217       notebook_page_close (win);
218     else {
219       win->is_quit = FALSE;
220       alert = tfe_alert_new_with_data ("Are␣you␣sure?", "Contents␣aren't␣saved␣
              yet.\nAre␣you␣sure␣to␣close?", "Close");
221       gtk_window_set_transient_for (GTK_WINDOW (alert), GTK_WINDOW (win));
222       g_signal_connect (TFE_ALERT (alert), "response", G_CALLBACK (alert_response_cb),
              win);
223       gtk_window_present (GTK_WINDOW (alert));
224     }
225   }
226
227   static void
228   new_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
229     TfeWindow *win = TFE_WINDOW (user_data);
230
231     tfe_window_notebook_page_new (win);
232   }
233
234   static void
235   saveas_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
236     TfeWindow *win = TFE_WINDOW (user_data);
```

```
237   TfeTextView *tv = get_current_textview (win->nb);
238
239   tfe_text_view_saveas (TFE_TEXT_VIEW (tv));
240 }
241
242 static void
243 pref_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data) {
244   TfeWindow *win = TFE_WINDOW (user_data);
245   GtkWidget *pref;
246
247   pref = tfe_pref_new ();
248   gtk_window_set_transient_for (GTK_WINDOW (pref), GTK_WINDOW (win));
249   gtk_window_present (GTK_WINDOW (pref));
250 }
251
252 static void
253 close_all_activated (GSimpleAction *action, GVariant *parameter, gpointer user_data)
      {
254   TfeWindow *win = TFE_WINDOW (user_data);
255
256   if (close_request_cb (win) == FALSE)
257     gtk_window_destroy (GTK_WINDOW (win));
258 }
259
260 /* --- public functions --- */
261
262 void
263 tfe_window_notebook_page_new (TfeWindow *win) {
264   GtkWidget *tv;
265   char *filename;
266
267   tv = tfe_text_view_new ();
268   filename = get_untitled ();
269   notebook_page_build (win, tv, filename);
270   g_free (filename);
271 }
272
273 void
274 tfe_window_notebook_page_new_with_files (TfeWindow *win, GFile **files, int n_files)
      {
275   int i;
276   GtkWidget *tv;
277   char *filename;
278
279   for (i = 0; i < n_files; i++)
280     if ((tv = tfe_text_view_new_with_file (*(files+i))) != NULL) {
281       filename = g_file_get_basename (*(files+i));
282       notebook_page_build (win, tv, filename);
283       g_free (filename);
284     }
285   if (gtk_notebook_get_n_pages (win->nb) == 0)
286     tfe_window_notebook_page_new (win);
287 }
288
289 static void
290 tfe_window_dispose (GObject *gobject) {
291   gtk_widget_dispose_template (GTK_WIDGET (gobject), TFE_TYPE_WINDOW);
292   G_OBJECT_CLASS (tfe_window_parent_class)->dispose (gobject);
293 }
294
295 static void
296 tfe_window_init (TfeWindow *win) {
297   GtkBuilder *build;
298   GMenuModel *menu;
```

```
299
300   gtk_widget_init_template (GTK_WIDGET (win));
301
302   build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfe/menu.ui");
303   menu = G_MENU_MODEL (gtk_builder_get_object (build, "menu"));
304   gtk_menu_button_set_menu_model (win->btnm, menu);
305   g_object_unref(build);
306
307 /* ----- action ----- */
308   const GActionEntry win_entries[] = {
309     { "open", open_activated, NULL, NULL, NULL },
310     { "save", save_activated, NULL, NULL, NULL },
311     { "close", close_activated, NULL, NULL, NULL },
312     { "new", new_activated, NULL, NULL, NULL },
313     { "saveas", saveas_activated, NULL, NULL, NULL },
314     { "pref", pref_activated, NULL, NULL, NULL },
315     { "close-all", close_all_activated, NULL, NULL, NULL }
316   };
317   g_action_map_add_action_entries (G_ACTION_MAP (win), win_entries, G_N_ELEMENTS
        (win_entries), win);
318
319   g_signal_connect (GTK_WINDOW (win), "close-request", G_CALLBACK
        (close_request_cb), NULL);
320 }
321
322 static void
323 tfe_window_class_init (TfeWindowClass *class) {
324   GObjectClass *object_class = G_OBJECT_CLASS (class);
325
326   object_class->dispose = tfe_window_dispose;
327   gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
        "/com/github/ToshioCP/tfe/tfewindow.ui");
328   gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeWindow, btnm);
329   gtk_widget_class_bind_template_child (GTK_WIDGET_CLASS (class), TfeWindow, nb);
330 }
331
332 GtkWidget *
333 tfe_window_new (GtkApplication *app) {
334   return GTK_WIDGET (g_object_new (TFE_TYPE_WINDOW, "application", app, NULL));
335 }
```

23 Pango, CSS and Application

23.1 PangoFontDescription

PangoFontDescription is a C structure for a font. You can get font family, style, weight and size. You can also get a string that includes font attributes. For example, suppose that the PangoFontDescription has a font of “Noto Sans Mono”, “Bold”, “Italic” and 12 points of size. Then the string converted from the PangoFontDescription is “Noto Sans Mono Bold Italic 12”.

- Font family is “Noto Sans Mono”.
- Font style is “Italic”.
- Font weight is “Bold”, or 700.
- Font size is 12 pt.

The font in CSS is different from the string from PangoFontDescription.

- `font: bold italic 12pt "Noto Sans Mono"`
- `Noto Sans Mono Bold Italic 12`

So, it may be easier to use each property, i.e. font-family, font-style, font-weight and font-size, to convert a PangoFontDescription data to CSS.

Refer to Pango document and W3C CSS Fonts Module Level 3 for further information.

23.2 Converter from PangoFontDescription to CSS

Two files `pfd2css.h` and `pfd2css.c` include the converter from PangoFontDescription to CSS.

```
#pragma once

#include <pango/pango.h>

// Pango font description to CSS style string
// Returned string is owned by the caller. The caller should free it when it becomes useless.

char*
pfd2css (PangoFontDescription *pango_font_desc);

// Each element (family, style, weight and size)

const char*
pfd2css_family (PangoFontDescription *pango_font_desc);

const char*
pfd2css_style (PangoFontDescription *pango_font_desc);

int
pfd2css_weight (PangoFontDescription *pango_font_desc);

// Returned string is owned by the caller. The caller should free it when it becomes useless.
char *
pfd2css_size (PangoFontDescription *pango_font_desc);
```

The five functions are public. The first function is a convenient function to set other four CSS at once.

```
#include <pango/pango.h>
#include "pfd2css.h"

// Pango font description to CSS style string
// Returned string is owned by caller. The caller should free it when it is useless.

char*
pfd2css (PangoFontDescription *pango_font_desc) {
  char *fontsize;

  fontsize = pfd2css_size (pango_font_desc);
  return g_strdup_printf ("font-family: \"%s\"; font-style: %s; font-weight: %d; font-size: %s;",
          pfd2css_family (pango_font_desc), pfd2css_style (pango_font_desc),
          pfd2css_weight (pango_font_desc), fontsize);
  g_free (fontsize);
}

// Each element (family, style, weight and size)

const char*
pfd2css_family (PangoFontDescription *pango_font_desc) {
  return pango_font_description_get_family (pango_font_desc);
}

const char*
pfd2css_style (PangoFontDescription *pango_font_desc) {
  PangoStyle pango_style = pango_font_description_get_style (pango_font_desc);
  switch (pango_style) {
  case PANGO_STYLE_NORMAL:
    return "normal";
  case PANGO_STYLE_ITALIC:
```

```
32      return "italic";
33    case PANGO_STYLE_OBLIQUE:
34      return "oblique";
35    default:
36      return "normal";
37    }
38  }
39
40  int
41  pfd2css_weight (PangoFontDescription *pango_font_desc) {
42    PangoWeight pango_weight = pango_font_description_get_weight (pango_font_desc);
43    switch (pango_weight) {
44    case PANGO_WEIGHT_THIN:
45      return 100;
46    case PANGO_WEIGHT_ULTRALIGHT:
47      return 200;
48    case PANGO_WEIGHT_LIGHT:
49      return 300;
50    case PANGO_WEIGHT_SEMILIGHT:
51      return 350;
52    case PANGO_WEIGHT_BOOK:
53      return 380;
54    case PANGO_WEIGHT_NORMAL:
55      return 400; /* or "normal" */
56    case PANGO_WEIGHT_MEDIUM:
57      return 500;
58    case PANGO_WEIGHT_SEMIBOLD:
59      return 600;
60    case PANGO_WEIGHT_BOLD:
61      return 700; /* or "bold" */
62    case PANGO_WEIGHT_ULTRABOLD:
63      return 800;
64    case PANGO_WEIGHT_HEAVY:
65      return 900;
66    case PANGO_WEIGHT_ULTRAHEAVY:
67      return 900; /* 1000 is available since CSS Fonts level 4 but GTK currently
          supports level 3. */
68    default:
69      return 400; /* "normal" */
70    }
71  }
72
73  char *
74  pfd2css_size (PangoFontDescription *pango_font_desc) {
75    if (pango_font_description_get_size_is_absolute (pango_font_desc))
76      return g_strdup_printf ("%dpx", pango_font_description_get_size
          (pango_font_desc) / PANGO_SCALE);
77    else
78      return g_strdup_printf ("%dpt", pango_font_description_get_size
          (pango_font_desc) / PANGO_SCALE);
79  }
```

- The function `pfd2css_family` returns font family.
- The function `pfd2css_style` returns font style which is one of “normal”, “italic” or “oblique”.
- The function `pfd2css_weight` returns font weight in integer. See the list below.
- The function `pfd2css_size` returns font size.
  - If the font description size is absolute, it returns the size of device unit, which is pixel. Otherwise the unit is point.
  - The function `pango_font_description_get_size` returns the integer of the size but it is multiplied by `PANGO_SCALE`. So, you need to divide it by `PANGO_SCALE`. The `PANGO_SCALE` is currently 1024, but this might be changed in the future. If the font size is 12pt, the size in pango is `12*PANGO_SCALE=12*1024=12288`.
- The function `pfd2css` returns a string of the font. For example, if a font “Noto Sans Mono Bold Italic

12” is given, it returns “font-family: Noto Sans Mono; font-style: italic; font-weight: 700; font-size: 12pt;”.

The font weight number is one of:

- 100 - Thin
- 200 - Extra Light (Ultra Light)
- 300 - Light
- 400 - Normal
- 500 - Medium
- 600 - Semi Bold (Demi Bold)
- 700 - Bold
- 800 - Extra Bold (Ultra Bold)
- 900 - Black (Heavy)

23.3 Application object

23.3.1 TfeApplication class

TfeApplication class is a child of GtkApplication. It has some instance variables. The header file defines the type macro and a public function.

```
1 #pragma once
2
3 #include <gtk/gtk.h>
4
5 #define TFE_TYPE_APPLICATION tfe_application_get_type ()
6 G_DECLARE_FINAL_TYPE (TfeApplication, tfe_application, TFE, APPLICATION,
      GtkApplication)
7
8 TfeApplication *
9 tfe_application_new (const char* application_id, GApplicationFlags flag);
```

The following code is extracted from `tfeapplication.c`. It builds TfeApplication class and instance.

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

struct _TfeApplication {
  GtkApplication parent;
  TfeWindow *win;
  GSettings *settings;
  GtkCssProvider *provider;
};

G_DEFINE_FINAL_TYPE (TfeApplication, tfe_application, GTK_TYPE_APPLICATION)

static void
tfe_application_dispose (GObject *gobject) {
  TfeApplication *app = TFE_APPLICATION (gobject);

  g_clear_object (&app->settings);
  g_clear_object (&app->provider);
  G_OBJECT_CLASS (tfe_application_parent_class)->dispose (gobject);
}

static void
tfe_application_init (TfeApplication *app) {
  app->settings = g_settings_new ("com.github.ToshioCP.tfe");
  g_signal_connect (app->settings, "changed::font-desc", G_CALLBACK
      (changed_font_cb), app);
  app->provider = gtk_css_provider_new ();
}

static void
```

```
tfe_application_class_init (TfeApplicationClass *class) {
  G_OBJECT_CLASS (class)->dispose = tfe_application_dispose;
  G_APPLICATION_CLASS (class)->startup = app_startup;
  G_APPLICATION_CLASS (class)->activate = app_activate;
  G_APPLICATION_CLASS (class)->open = app_open;
}

TfeApplication *
tfe_application_new (const char* application_id, GApplicationFlags flag) {
  return TFE_APPLICATION (g_object_new (TFE_TYPE_APPLICATION, "application-id",
      application_id, "flags", flag, NULL));
}
```

- The structure `_TfeApplication` is defined. It has four members. One is its parents and the others are kinds of instance variables. The members are usually initialized in the instance initialization function. And they are released in the disposal function or freed in the finalization function. The members are:
  - win: main window instance
  - settings: GSettings instance.it is bound to "font-desc" item in the GSettings
  - provider: a provider for the font of the textview.
- The macro `G_DEFINE_FINAL_TYPE` defines `tfe_application_get_type` function and some other useful things.
- The function `tfe_application_class_init` initializes the TfeApplication class. It overrides four class methods. Three class methods `startup`, `activate` and `open` points the default signal handlers. The overriding changes the default handlers. You can connect the handlers with `g_signal_connect` macro but the result is different. The macro connects a user handler to the signal. The default handler still exists and no change is made to them.
- The function `tfe_application_init` initializes an instance.
  - Creates a new GSettings instance and make `app->settings` point it. Then connects the handler `changed_font_cb` to the "changed::font-desc" signal.
  - Creates a new GtkCssProvider instance and make `app->provider` point it.
- The function `tfe_application_dispose` releases the GSettings and GtkCssProvider instances. Then, call the parent's dispose handler. It is called "chaining up". See GObject document.

23.3.2 Startup signal handlers

```
1  static void
2  app_startup (GApplication *application) {
3    TfeApplication *app = TFE_APPLICATION (application);
4    int i;
5    GtkCssProvider *provider = gtk_css_provider_new ();
6    GdkDisplay *display;
7
8    G_APPLICATION_CLASS (tfe_application_parent_class)->startup (application);
9
10   app->win = TFE_WINDOW (tfe_window_new (GTK_APPLICATION (app)));
11
12   gtk_css_provider_load_from_data (provider, "textview␣{padding:␣10px;}", -1);
13   display = gdk_display_get_default ();
14   gtk_style_context_add_provider_for_display (display, GTK_STYLE_PROVIDER (provider),
15                                               GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
16   g_object_unref (provider);
17   gtk_style_context_add_provider_for_display (display, GTK_STYLE_PROVIDER
         (app->provider),
18                                               GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
19
20   changed_font_cb (app->settings, "font-desc", app); // Sets the text view font to
         the font from the gsettings data base.
21
22 /* ----- accelerator ----- */
23   struct {
24     const char *action;
25     const char *accels[2];
26   } action_accels[] = {
```

```
27      { "win.open", { "<Control>o", NULL } },
28      { "win.save", { "<Control>s", NULL } },
29      { "win.close", { "<Control>w", NULL } },
30      { "win.new", { "<Control>n", NULL } },
31      { "win.saveas", { "<Shift><Control>s", NULL } },
32      { "win.close-all", { "<Control>q", NULL } },
33    };
34
35    for (i = 0; i < G_N_ELEMENTS(action_accels); i++)
36      gtk_application_set_accels_for_action(GTK_APPLICATION(app),
            action_accels[i].action, action_accels[i].accels);
37  }
```

The function `app_startup` replace the default signal handlers. It does five things.

- Calls the parent's startup handler. It is called "chaining up". The "startup" default handler runs before user handlers. So the call for the parent's handler must be done at the beginning.
- Creates the main window. This application has only one top level window. In that case, it is a good way to create the window in the startup handler, which is called only once. Activate or open handlers can be called twice or more. Therefore, if you create a window in the activate or open handler, two or more windows can be created.
- Sets the default display CSS to "textview {padding: 10px;}". It sets the GtkTextView, or TfeTextView, padding to 10px and makes the text easier to read. This CSS is fixed and never changed through the application life.
- Adds another CSS provider, which is pointed by `app->provider`, to the default display. This CSS depends on the GSettings "font-desc" value and it can be changed during the application life time. And calls `changed_font_cb` to update the font CSS setting.
- Sets application accelerator with the function `gtk_application_set_accels_for_action`. Accelerators are kinds of short cut key functions. For example, `Ctrl+O` is an accelerator to activate "open" action. Accelerators are written in the array `action-accels[]`. Its element is a structure `struct {const char *action; const char *accels[2];}`. The member `action` is an action name. The member `accels` is an array of two pointers. For example, `{"win.open", { "<Control>o", NULL }}` tells that the accelerator `Ctrl+O` is connected to the "win.open" action. The second element of `accels` is NULL which is the end mark. You can define more than one accelerator keys and the list must ends with NULL (zero). If you want to do so, the array length needs to be three or more. For example, `{"win.open", { "<Control>o", "<Alt>o", NULL }}` means two accelerators `Ctrl+O` and `Alt+O` is connected to the "win.open" action. The parser recognizes "<control>o", "<Shift><Alt>F2", "<Ctrl>minus" and so on. If you want to use symbol key like "<Ctrl>-", use "<Ctrl>minus" instead. Such relation between lower case and symbol (character code) is specified in `gdkkeysyms.h` in the GTK 4 source code.

23.3.3 Activate and open signal handlers

Two functions `app_activate` and `app_open` replace the default signal handlers.

```
1   static void
2   app_activate (GApplication *application) {
3     TfeApplication *app = TFE_APPLICATION (application);
4
5     tfe_window_notebook_page_new (app->win);
6     gtk_window_present (GTK_WINDOW (app->win));
7   }
8
9   static void
10  app_open (GApplication *application, GFile ** files, gint n_files, const gchar
        *hint) {
11    TfeApplication *app = TFE_APPLICATION (application);
12
13    tfe_window_notebook_page_new_with_files (app->win, files, n_files);
14    gtk_window_present (GTK_WINDOW (app->win));
15  }
```

The original default handlers don't do useful works and you don't need to chain up to the parent's default handlers. They just create notebook pages and show the top level window.

23.3.4 CSS font setting

```
static void
changed_font_cb (GSettings *settings, char *key, gpointer user_data) {
  TfeApplication *app = TFE_APPLICATION (user_data);
  char *font, *s, *css;
  PangoFontDescription *pango_font_desc;

  if (g_strcmp0(key, "font-desc") != 0)
    return;
  font = g_settings_get_string (app->settings, "font-desc");
  pango_font_desc = pango_font_description_from_string (font);
  g_free (font);
  s = pfd2css (pango_font_desc); // converts Pango Font Description into CSS style
      string
  pango_font_description_free (pango_font_desc);
  css = g_strdup_printf ("textview␣{%s}", s);
  gtk_css_provider_load_from_data (app->provider, css, -1);
  g_free (s);
  g_free (css);
}
```

The function `changed_font_cb` is a handler for “changed::font-desc” signal on the GSettings instance. The signal name is “changed” and “font-desc” is a key name. This signal is emitted when the value of the “font-desc” key is changed. The value is bound to the “font-desc” property of the GtkFontDialogButton instance. Therefore, the handler `changed_font_cb` is called when the user selects and updates a font through the font dialog.

A string is retrieved from the GSetting database and converts it into a pango font description. And a CSS string is made by the function `pfd2css` and `g_strdup_printf`. Then the css provider is set to the string. The provider has been inserted to the current display in advance. So, the font is applied to the display.

23.4 Other files

main.c

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

#define APPLICATION_ID "com.github.ToshioCP.tfe"

int
main (int argc, char **argv) {
  TfeApplication *app;
  int stat;

  app = tfe_application_new (APPLICATION_ID, G_APPLICATION_HANDLES_OPEN);
  stat =g_application_run (G_APPLICATION (app), argc, argv);
  g_object_unref (app);
  return stat;
}
```

Resource XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/com/github/ToshioCP/tfe">
    <file>tfewindow.ui</file>
    <file>tfepref.ui</file>
    <file>tfealert.ui</file>
    <file>menu.ui</file>
  </gresource>
</gresources>
```

GSchema XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<schemalist>
  <schema path="/com/github/ToshioCP/tfe/" id="com.github.ToshioCP.tfe">
    <key name="font-desc" type="s">
      <default>'Monospace 12'</default>
      <summary>Font</summary>
      <description>A font to be used for textview.</description>
    </key>
  </schema>
</schemalist>
```

Meson.build

```
project('tfe', 'c', license : 'GPL-3.0-or-later', meson_version:'>=1.0.1', version:
    '0.5')

gtkdep = dependency('gtk4')

gnome = import('gnome')
resources = gnome.compile_resources('resources','tfe.gresource.xml')
gnome.compile_schemas(depend_files: 'com.github.ToshioCP.tfe.gschema.xml')

sourcefiles = files('main.c', 'tfeapplication.c', 'tfewindow.c', 'tfepref.c',
    'tfealert.c', 'pfd2css.c', '../tfetextview/tfetextview.c')

executable(meson.project_name(), sourcefiles, resources, dependencies: gtkdep,
    export_dynamic: true, install: true)

schema_dir = get_option('prefix') / get_option('datadir') / 'glib-2.0/schemas/'
install_data('com.github.ToshioCP.tfe.gschema.xml', install_dir: schema_dir)
gnome.post_install (glib_compile_schemas: true)
```

- The function `project` defines project and initialize meson. The first argument is the project name and the second is the language name. The other arguments are keyword arguments.
- The function `dependency` defines the denpendent library. Tfe depends GTK4. This is used to create `pkg-config` option in the command line of C compiler to include header files and link libraries. The returned object `gtkdep` is used as an argument to the `executable` function later.
- The function `import` imports an extension module. The GNOME module has some convenient methods like `gnome.compile_resources` and `gnome.compile_schemas`.
- The method `gnome.compile_resources` compiles and creates resource files. The first argument is the resource name without extension and the second is the name of XML file. The returned value is an array `['resources.c', 'resources.h']`.
- The function `gnome.compile_schemas` compiles the schema files in the current directory. This just creates `gschemas.compiled` in the build directory. It is used to test the executable binary in the build directory. The function doesn't install the schema file.
- The function `files` creates a File Object.
- The function `executable` defines the compilation elements such as target name, source files, dependencies and installation. The target name is "tfe". The source files are elements of 'sourcefiles' and 'resources'. It uses GTK4 libraries. It can be installed.
- The last three lines are post install work. The variable `schema_dir` is the directory stored the schema file. If meson runs with `--prefix=$HOME/.local` argument, it is `$HOME/.local/share/glib-2.9/schemas`. The function `install_data` copies the first argument file into the second argument directory. The method `gnome.post_install` runs `glib-compile-schemas` and updates `gschemas_compiled` file.

23.5 Compilation and installation.

If you want to install it to your local area, use `--prefix=$HOME/.local` or `--prefix=$HOME` option. If you want to install it to the system area, no option is needed. It will be installed under `/user/local` directory.

```
$ meson setup --prefix=$HOME/.local _build
$ ninja -C _build
$ ninja -C _build install
```

You need root privilege to install it to the system area..

```
$ meson setup _build
$ ninja -C _build
$ sudo ninja -C _build install
```

Source files are in src/tfe6 directory.

We made a very small text editor. You can add features to this editor. When you add a new feature, be careful about the structure of the program. Maybe you need to divide a file into several files. It isn't good to put many things into one file. And it is important to think about the relationship between source files and widget structures.

The source files are in the Gtk4 tutorial GitHub repository. Download it and see `src/tfe6` directory.

Note: When the menu button is clicked, error messages are printed.

```
(tfe:31153): Gtk-CRITICAL **: 13:05:40.746: _gtk_css_corner_value_get_x: assertion
    'corner->class == &GTK_CSS_VALUE_CORNER' failed
```

I found a message in the GNOME Discourse. The comment says that GTK 4.10 has a bug and it is fixed in the version 4.10.5. I haven't check 4.10.5 yet, where the UBUNTU GTK4 is still 4.10.4.

24 GtkDrawingArea and Cairo

If you want to draw shapes or paint images dynamically on the screen, use the GtkDrawingArea widget.

GtkDrawingArea provides a cairo drawing context. You can draw images with cairo library functions. This section describes:

1. Cairo, but briefly
2. GtkDrawingArea, with a very simple example.

24.1 Cairo

Cairo is a drawing library for two dimensional graphics. There are a lot of documents on Cairo's website. If you aren't familiar with Cairo, it is worth reading the tutorial.

The following is an introduction to the Cairo library. First, you need to know surfaces, sources, masks, destinations, cairo context and transformations.

- A surface represents an image. It is like a canvas. We can draw shapes and images with different colors on surfaces.
- The source pattern, or simply source, is like paint, which will be transferred to destination surface by cairo functions.
- The mask describes the area to be used in the copy;
- The destination is a target surface;
- The cairo context manages the transfer from source to destination, through mask with its functions; For example, `cairo_stroke` is a function to draw a path to the destination by the transfer.
- A transformation can be applied before the transfer completes. The transformation which is applied is called affine, which is a mathematical term meaning transofrmations that preserve straight lines. Scaling, rotating, reflecting, shearing and translating are examples of affine transformations. They are mathematically represented by matrix multiplication and vector addition. In this section we don't use it, instead we will only use the identity transformation. This means that the coordinates in the source and mask are the same as the coordinates in destination.

The instruction is as follows:

1. Create a surface. This will be the destination.
2. Create a cairo context with the surface, the surface will be the destination of the context.
3. Create a source pattern within the context.
4. Create paths, which are lines, rectangles, arcs, texts or more complicated shapes in the mask.
5. Use a drawing operator such as `cairo_stroke` to transfer the paint in the source to the destination.
6. Save the destination surface to a file if necessary.

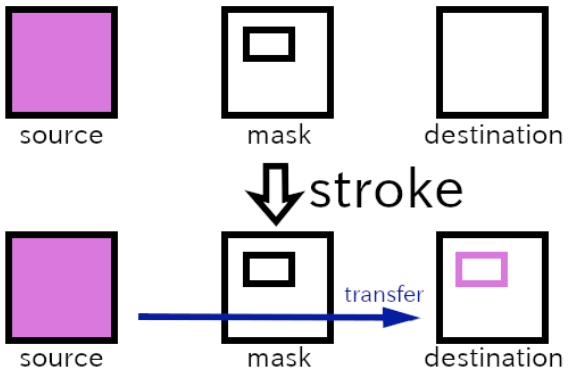


Figure 33: Stroke a rectangle

Here's a simple example program that draws a small square and saves it as a png file. The path of the file is `src/misc/cairo.c`.

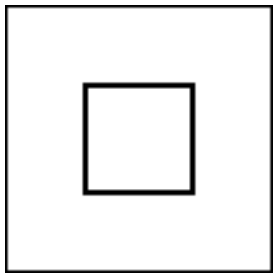
```
1  #include <cairo.h>
2
3  int
4  main (int argc, char **argv)
5  {
6    cairo_surface_t *surface;
7    cairo_t *cr;
8    int width = 100;
9    int height = 100;
10   int square_size = 40.0;
11
12   /* Create surface and cairo */
13   surface = cairo_image_surface_create (CAIRO_FORMAT_RGB24, width, height);
14   cr = cairo_create (surface);
15
16   /* Drawing starts here. */
17   /* Paint the background white */
18   cairo_set_source_rgb (cr, 1.0, 1.0, 1.0);
19   cairo_paint (cr);
20   /* Draw a black rectangle */
21   cairo_set_source_rgb (cr, 0.0, 0.0, 0.0);
22   cairo_set_line_width (cr, 2.0);
23   cairo_rectangle (cr,
24                    width/2.0 - square_size/2,
25                    height/2.0 - square_size/2,
26                    square_size,
27                    square_size);
28   cairo_stroke (cr);
29
30   /* Write the surface to a png file and clean up cairo and surface. */
31   cairo_surface_write_to_png (surface, "rectangle.png");
32   cairo_destroy (cr);
33   cairo_surface_destroy (surface);
34
35   return 0;
36 }
```

- 1: Includes the header file of Cairo.
- 6: `cairo_surface_t` is the type of a surface.
- 7: `cairo_t` is the type of a cairo context.
- 8-10: `width` and `height` are the size of `surface`. `square_size` is the size of a square to be drawn on the surface.

- 13: `cairo_image_surface_create` creates an image surface. `CAIRO_FORMAT_RGB24` is a constant which means that each pixel has red, green and blue data, and each data point is an 8 bits number (for 24 bits in total). Modern displays have this type of color depth. Width and height are in pixels and given as integers.
- 14: Creates cairo context. The surface given as an argument will be the destination of the context.
- 18: `cairo_set_source_rgb` creates a source pattern, which is a solid white paint. The second to fourth arguments are red, green and blue color values respectively, and they are of type float. The values are between zero (0.0) and one (1.0). Black is (0.0,0.0,0.0) and white is (1.0,1.0,1.0).
- 19: `cairo_paint` copies everywhere in the source to destination. The destination is filled with white pixels with this command.
- 21: Sets the source color to black.
- 22: `cairo_set_line_width` sets the width of lines. In this case, the line width is set to be two pixels and will end up that same size. (It is because the transformation is identity. If the transformation isn't identity, for example scaling with the factor three, the actual width in destination will be six (2x3=6) pixels.)
- 23: Draws a rectangle (square) on the mask. The square is located at the center.
- 24: `cairo_stroke` transfers the source to destination through the rectangle in the mask.
- 31: Outputs the image to a png file `rectangle.png`.
- 32: Destroys the context. At the same time the source is destroyed.
- 33: Destroys the surface.

To compile this, change your current directory to `src/misc` and type the following.

```
$ gcc `pkg-config --cflags cairo` cairo.c `pkg-config --libs cairo`
```


s

See the Cairo's website for further information.

24.2 GtkDrawingArea

The following is a very simple example.

```
1  #include <gtk/gtk.h>
2
3  static void
4  draw_function (GtkDrawingArea *area, cairo_t *cr, int width, int height, gpointer
       user_data) {
5    int square_size = 40.0;
6
7    cairo_set_source_rgb (cr, 1.0, 1.0, 1.0); /* white */
8    cairo_paint (cr);
9    cairo_set_line_width (cr, 2.0);
10   cairo_set_source_rgb (cr, 0.0, 0.0, 0.0); /* black */
11   cairo_rectangle (cr,
12                    width/2.0 - square_size/2,
13                    height/2.0 - square_size/2,
14                    square_size,
15                    square_size);
16   cairo_stroke (cr);
17 }
18
19 static void
20 app_activate (GApplication *app, gpointer user_data) {
21   GtkWidget *win = gtk_application_window_new (GTK_APPLICATION (app));
```

```
22    GtkWidget *area = gtk_drawing_area_new ();
23
24    gtk_window_set_title (GTK_WINDOW (win), "da1");
25    gtk_drawing_area_set_draw_func (GTK_DRAWING_AREA (area), draw_function, NULL,
          NULL);
26    gtk_window_set_child (GTK_WINDOW (win), area);
27
28    gtk_window_present (GTK_WINDOW (win));
29  }
30
31  #define APPLICATION_ID "com.github.ToshioCP.da1"
32
33  int
34  main (int argc, char **argv) {
35    GtkApplication *app;
36    int stat;
37
38    app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
39    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
40    stat =g_application_run (G_APPLICATION (app), argc, argv);
41    g_object_unref (app);
42    return stat;
43  }
```

The function `main` is almost same as before. The two functions `app_activate` and `draw_function` are important in this example.

- 22: Creates a GtkDrawingArea instance.
- 25: Sets a drawing function of the widget. GtkDrawingArea widget uses the function `draw_function` to draw the contents of itself whenever its necessary. For example, when a user drag a mouse pointer and resize a top-level window, GtkDrawingArea also changes the size. Then, the whole window needs to be redrawn. For the information of `gtk_drawing_area_set_draw_func`, see Gtk API Reference – gtk_drawing_area_set_draw_func.

The drawing function has five parameters.

```
void drawing_function (GtkDrawingArea *drawing_area, cairo_t *cr, int width, int
    height,
                       gpointer user_data);
```

The first parameter is the GtkDrawingArea widget. You can't change any properties, for example `content-width` or `content-height`, in this function. The second parameter is a cairo context given by the widget. The destination surface of the context is connected to the contents of the widget. What you draw to this surface will appear in the widget on the screen. The third and fourth parameters are the size of the destination surface. Now, look at the program again.

- 3-17: The drawing function.
- 7-8: Sets the source to be white and paint the destination white.
- 9: Sets the line width to be 2.
- 10: Sets the source to be black.
- 11-15: Adds a rectangle to the mask.
- 16: Draws the rectangle with black color to the destination.

The program is src/misc/da1.c. Compile and run it, then a window with a black rectangle (square) appears. Try resizing the window. The square always appears at the center of the window because the drawing function is invoked each time the window is resized.

25 Periodic Events

This chapter was written by Paul Schulz paul@mawsonlakes.org.

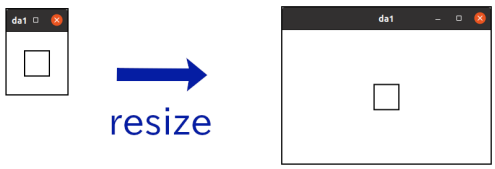


Figure 34: Square in the window

25.1 How do we create an animation?

In this section we will continue to build on our previous work. We will create an analog clock application. By adding a function which periodically redraws GtkDrawingArea, the clock will be able to continuously display the time.

The application uses a compiled in 'resource' file, so if the GTK4 libraries and their dependencies are installed and available, the application will run from anywhere.

The program also makes use of some standard mathematical and time handling functions.

The clocks mechanics were taken from a Cairo drawing example, using gtkmm4, which can be found here.

The complete code is at the end.

25.2 Drawing the clock face, hour, minute and second hands

The `draw_clock()` function does all the work. See the in-file comments for an explanation of how the Cairo drawing works.

For a detailed reference of what each of the Cairo functions does see the cairo_t reference.

```
1  static void
2  draw_clock (GtkDrawingArea *area, cairo_t *cr, int width, int height, gpointer
       user_data) {
3
4      // Scale to unit square and translate (0, 0) to be (0.5, 0.5), i.e.
5      // the center of the window
6      cairo_scale(cr, width, height);
7      cairo_translate(cr, 0.5, 0.5);
8
9      // Set the line width and save the cairo drawing state.
10     cairo_set_line_width(cr, m_line_width);
11     cairo_save(cr);
12
13     // Set the background to a slightly transparent green.
14     cairo_set_source_rgba(cr, 0.337, 0.612, 0.117, 0.9);   // green
15     cairo_paint(cr);
16
17     // Resore back to precious drawing state and draw the circular path
18     // representing the clockface. Save this state (including the path) so we
19     // can reuse it.
20     cairo_restore(cr);
21     cairo_arc(cr, 0.0, 0.0, m_radius, 0.0, 2.0 * M_PI);
22     cairo_save(cr);
23
24     // Fill the clockface with white
25     cairo_set_source_rgba(cr, 1.0, 1.0, 1.0, 0.8);
26     cairo_fill_preserve(cr);
27     // Restore the path, paint the outside of the clock face.
28     cairo_restore(cr);
29     cairo_stroke_preserve(cr);
30     // Set the 'clip region' to the inside of the path (fill region).
```

```
31      cairo_clip(cr);
32
33      // Clock ticks
34      for (int i = 0; i < 12; i++)
35      {
36          // Major tick size
37          double inset = 0.05;
38
39          // Save the graphics state, restore after drawing tick to maintain pen
40          // size
41          cairo_save(cr);
42          cairo_set_line_cap(cr, CAIRO_LINE_CAP_ROUND);
43
44          // Minor ticks are shorter, and narrower.
45          if(i % 3 != 0)
46          {
47              inset *= 0.8;
48              cairo_set_line_width(cr, 0.03);
49          }
50
51          // Draw tick mark
52          cairo_move_to(
53              cr,
54              (m_radius - inset) * cos (i * M_PI / 6.0),
55              (m_radius - inset) * sin (i * M_PI / 6.0));
56          cairo_line_to(
57              cr,
58              m_radius * cos (i * M_PI / 6.0),
59              m_radius * sin (i * M_PI / 6.0));
60          cairo_stroke(cr);
61          cairo_restore(cr); /* stack-pen-size */
62      }
63
64      // Draw the analog hands
65
66      // Get the current Unix time, convert to the local time and break into time
67      // structure to read various time parts.
68      time_t rawtime;
69      time(&rawtime);
70      struct tm * timeinfo = localtime (&rawtime);
71
72      // Calculate the angles of the hands of our clock
73      double hours   = timeinfo->tm_hour * M_PI / 6.0;
74      double minutes = timeinfo->tm_min * M_PI / 30.0;
75      double seconds = timeinfo->tm_sec * M_PI / 30.0;
76
77      // Save the graphics state
78      cairo_save(cr);
79      cairo_set_line_cap(cr, CAIRO_LINE_CAP_ROUND);
80
81      cairo_save(cr);
82
83      // Draw the seconds hand
84      cairo_set_line_width(cr, m_line_width / 3.0);
85      cairo_set_source_rgba(cr, 0.7, 0.7, 0.7, 0.8);   // gray
86      cairo_move_to(cr, 0.0, 0.0);
87      cairo_line_to(cr,
88                    sin(seconds) * (m_radius * 0.9),
89                    -cos(seconds) * (m_radius * 0.9));
90      cairo_stroke(cr);
91      cairo_restore(cr);
92
93      // Draw the minutes hand
94      cairo_set_source_rgba(cr, 0.117, 0.337, 0.612, 0.9);   // blue
```

```
95      cairo_move_to(cr, 0, 0);
96      cairo_line_to(cr,
97                    sin(minutes + seconds / 60) * (m_radius * 0.8),
98                    -cos(minutes + seconds / 60) * (m_radius * 0.8));
99      cairo_stroke(cr);
100
101     // draw the hours hand
102     cairo_set_source_rgba(cr, 0.337, 0.612, 0.117, 0.9);   // green
103     cairo_move_to(cr, 0.0, 0.0);
104     cairo_line_to(cr,
105                   sin(hours + minutes / 12.0) * (m_radius * 0.5),
106                   -cos(hours + minutes / 12.0) * (m_radius * 0.5));
107     cairo_stroke(cr);
108     cairo_restore(cr);
109
110     // Draw a little dot in the middle
111     cairo_arc(cr, 0.0, 0.0, m_line_width / 3.0, 0.0, 2.0 * M_PI);
112     cairo_fill(cr);
113 }
```

In order for the clock to be drawn, the drawing function `draw_clock()` needs to be registered with GTK4. This is done in the `app_activate()` function (on line 24).

Whenever the application needs to redraw the GtkDrawingArea, it will now call `draw_clock()`.

There is still a problem though. In order to animate the clock we need to also tell the application that the clock needs to be redrawn every second. This process starts by registering (on the next line, line 15) a timeout function with `g_timeout_add()` that will wakeup and run another function `time_handler`, every second (or 1000ms).

```
1  static void
2  app_activate (GApplication *app, gpointer user_data) {
3      GtkWidget *win;
4      GtkWidget *clock;
5      GtkBuilder *build;
6
7      build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfc/tfc.ui");
8      win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
9      gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
10
11     clock = GTK_WIDGET (gtk_builder_get_object (build, "clock"));
12     g_object_unref(build);
13
14     gtk_drawing_area_set_draw_func(GTK_DRAWING_AREA (clock), draw_clock, NULL, NULL);
15     g_timeout_add(1000, (GSourceFunc) time_handler, (gpointer) clock);
16     gtk_window_present(GTK_WINDOW (win));
17
18 }
```

Our `time_handler()` function is very simple, as it just calls `gtk_widget_queue_draw()` which schedules a redraw of the widget.

```
1  gboolean
2  time_handler(GtkWidget* widget) {
3      gtk_widget_queue_draw(widget);
4
5      return TRUE;
6  }
```

.. and that is all there is to it. If you compile and run the example you will get a ticking analog clock.

If you get this working, you can try modifying some of the code in `draw_clock()` to tweak the application (such as change the color or size and length of the hands) or even add text, or create a digital clock.

25.3 The Complete code

You can find the source files in the `tfc` directory. it can be compiled with `./comp tfc`.

`tfc.c`

```
1  #include <gtk/gtk.h>
2  #include <math.h>
3  #include <time.h>
4
5  float m_radius     = 0.42;
6  float m_line_width = 0.05;
7
8  static void
9  draw_clock (GtkDrawingArea *area, cairo_t *cr, int width, int height, gpointer
       user_data) {
10
11     // Scale to unit square and translate (0, 0) to be (0.5, 0.5), i.e.
12     // the center of the window
13     cairo_scale(cr, width, height);
14     cairo_translate(cr, 0.5, 0.5);
15
16     // Set the line width and save the cairo drawing state.
17     cairo_set_line_width(cr, m_line_width);
18     cairo_save(cr);
19
20     // Set the background to a slightly transparent green.
21     cairo_set_source_rgba(cr, 0.337, 0.612, 0.117, 0.9);   // green
22     cairo_paint(cr);
23
24     // Resore back to precious drawing state and draw the circular path
25     // representing the clockface. Save this state (including the path) so we
26     // can reuse it.
27     cairo_restore(cr);
28     cairo_arc(cr, 0.0, 0.0, m_radius, 0.0, 2.0 * M_PI);
29     cairo_save(cr);
30
31     // Fill the clockface with white
32     cairo_set_source_rgba(cr, 1.0, 1.0, 1.0, 0.8);
33     cairo_fill_preserve(cr);
34     // Restore the path, paint the outside of the clock face.
35     cairo_restore(cr);
36     cairo_stroke_preserve(cr);
37     // Set the 'clip region' to the inside of the path (fill region).
38     cairo_clip(cr);
39
40     // Clock ticks
41     for (int i = 0; i < 12; i++)
42     {
43         // Major tick size
44         double inset = 0.05;
45
46         // Save the graphics state, restore after drawing tick to maintain pen
47         // size
48         cairo_save(cr);
49         cairo_set_line_cap(cr, CAIRO_LINE_CAP_ROUND);
50
51         // Minor ticks are shorter, and narrower.
52         if(i % 3 != 0)
53         {
54             inset *= 0.8;
55             cairo_set_line_width(cr, 0.03);
56         }
57
58         // Draw tick mark
```

```
59          cairo_move_to(
60              cr,
61              (m_radius - inset) * cos (i * M_PI / 6.0),
62              (m_radius - inset) * sin (i * M_PI / 6.0));
63          cairo_line_to(
64              cr,
65              m_radius * cos (i * M_PI / 6.0),
66              m_radius * sin (i * M_PI / 6.0));
67          cairo_stroke(cr);
68          cairo_restore(cr); /* stack-pen-size */
69      }
70
71      // Draw the analog hands
72
73      // Get the current Unix time, convert to the local time and break into time
74      // structure to read various time parts.
75      time_t rawtime;
76      time(&rawtime);
77      struct tm * timeinfo = localtime (&rawtime);
78
79      // Calculate the angles of the hands of our clock
80      double hours   = timeinfo->tm_hour * M_PI / 6.0;
81      double minutes = timeinfo->tm_min * M_PI / 30.0;
82      double seconds = timeinfo->tm_sec * M_PI / 30.0;
83
84      // Save the graphics state
85      cairo_save(cr);
86      cairo_set_line_cap(cr, CAIRO_LINE_CAP_ROUND);
87
88      cairo_save(cr);
89
90      // Draw the seconds hand
91      cairo_set_line_width(cr, m_line_width / 3.0);
92      cairo_set_source_rgba(cr, 0.7, 0.7, 0.7, 0.8);   // gray
93      cairo_move_to(cr, 0.0, 0.0);
94      cairo_line_to(cr,
95                    sin(seconds) * (m_radius * 0.9),
96                    -cos(seconds) * (m_radius * 0.9));
97      cairo_stroke(cr);
98      cairo_restore(cr);
99
100     // Draw the minutes hand
101     cairo_set_source_rgba(cr, 0.117, 0.337, 0.612, 0.9);   // blue
102     cairo_move_to(cr, 0, 0);
103     cairo_line_to(cr,
104                   sin(minutes + seconds / 60) * (m_radius * 0.8),
105                   -cos(minutes + seconds / 60) * (m_radius * 0.8));
106     cairo_stroke(cr);
107
108     // draw the hours hand
109     cairo_set_source_rgba(cr, 0.337, 0.612, 0.117, 0.9);   // green
110     cairo_move_to(cr, 0.0, 0.0);
111     cairo_line_to(cr,
112                   sin(hours + minutes / 12.0) * (m_radius * 0.5),
113                   -cos(hours + minutes / 12.0) * (m_radius * 0.5));
114     cairo_stroke(cr);
115     cairo_restore(cr);
116
117     // Draw a little dot in the middle
118     cairo_arc(cr, 0.0, 0.0, m_line_width / 3.0, 0.0, 2.0 * M_PI);
119     cairo_fill(cr);
120 }
121
122
```

```
123 gboolean
124 time_handler(GtkWidget* widget) {
125     gtk_widget_queue_draw(widget);
126 
127     return TRUE;
128 }
129 
130 
131 static void
132 app_activate (GApplication *app, gpointer user_data) {
133     GtkWidget *win;
134     GtkWidget *clock;
135     GtkBuilder *build;
136 
137     build = gtk_builder_new_from_resource ("/com/github/ToshioCP/tfc/tfc.ui");
138     win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
139     gtk_window_set_application (GTK_WINDOW (win), GTK_APPLICATION (app));
140 
141     clock = GTK_WIDGET (gtk_builder_get_object (build, "clock"));
142     g_object_unref(build);
143 
144     gtk_drawing_area_set_draw_func(GTK_DRAWING_AREA (clock), draw_clock, NULL, NULL);
145     g_timeout_add(1000, (GSourceFunc) time_handler, (gpointer) clock);
146     gtk_window_present(GTK_WINDOW (win));
147 
148 }
149 
150 static void
151 app_open (GApplication *app, GFile **files, gint n_files, gchar *hint, gpointer
        user_data) {
152     app_activate(app,user_data);
153 }
154 
155 int
156 main (int argc, char **argv) {
157     GtkApplication *app;
158     int stat;
159 
160     app = gtk_application_new ("com.github.ToshioCP.tfc",
            G_APPLICATION_HANDLES_OPEN);
161     g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
162     g_signal_connect (app, "open", G_CALLBACK (app_open), NULL);
163     stat = g_application_run (G_APPLICATION (app), argc, argv);
164     g_object_unref (app);
165     return stat;
166 }
```

tfc.ui

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <object class="GtkApplicationWindow" id="win">
4      <property name="title">Clock</property>
5      <property name="default-width">200</property>
6      <property name="default-height">200</property>
7      <child>
8        <object class="GtkDrawingArea" id="clock">
9          <property name="hexpand">TRUE</property>
10         <property name="vexpand">TRUE</property>
11       </object>
12     </child>
13   </object>
14 </interface>
```

tfc.gresource.xml

Figure 35: down the button

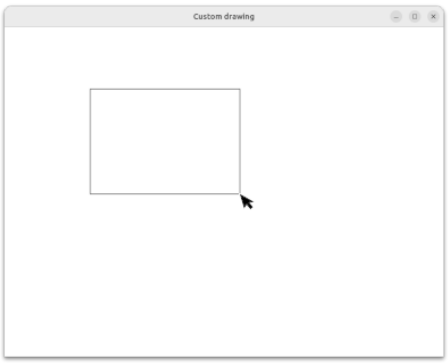


Figure 36: Move the mouse

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/com/github/ToshioCP/tfc">
    <file>tfc.ui</file>
  </gresource>
</gresources>
```

comp

```
glib-compile-resources $1.gresource.xml --target=$1.gresource.c --generate-source
gcc `pkg-config --cflags gtk4` $1.gresource.c $1.c `pkg-config --libs gtk4` -lm
```

26 Custom drawing

Custom drawing is to draw shapes dynamically. This section shows an example of custom drawing. You can draw rectangles by dragging the mouse.

Down the button.

Move the mouse

Up the button.

The programs are at `src/custom_drawing` directory. Download the repository and see the directory. There are four files.

- meson.build
- rect.c
- rect.gresource.xml
- rect.ui

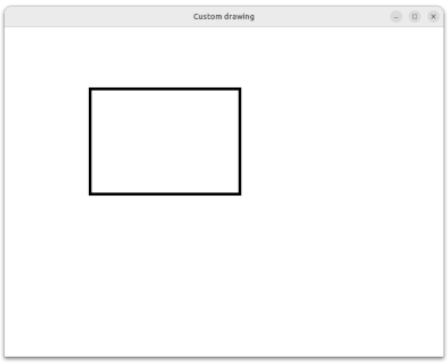


Figure 37: Up the button

26.1 rect.gresource.xml

This file describes a ui file to compile. The compiler glib-compile-resources uses it.

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/com/github/ToshioCP/rect">
    <file>rect.ui</file>
  </gresource>
</gresources>
```

The prefix is `/com/github/ToshioCP/rect` and the file is `rect.ui`. Therefore, GtkBuilder reads the resource from `/com/github/ToshioCP/rect/rect.ui`.

26.2 rect.ui

The following is the ui file that defines the widgets. There are two widgets which are GtkApplicationWindow and GtkDrawingArea. The ids are `win` and `da` respectively.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <object class="GtkApplicationWindow" id="win">
    <property name="default-width">800</property>
    <property name="default-height">600</property>
    <property name="resizable">FALSE</property>
    <property name="title">Custom drawing</property>
    <child>
      <object class="GtkDrawingArea" id="da">
        <property name="hexpand">TRUE</property>
        <property name="vexpand">TRUE</property>
      </object>
    </child>
  </object>
</interface>
```

26.3 rect.c

26.3.1 GtkApplication

This program uses GtkApplication. The application ID is `com.github.ToshioCP.rect`.

```
#define APPLICATION_ID "com.github.ToshioCP.rect"
```

See GNOME Developer Documentation for further information.

The function `main` is called at the beginning of the application.

```
int
main (int argc, char **argv) {
```

```
3   GtkApplication *app;
4   int stat;
5
6   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_HANDLES_OPEN);
7   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
8   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
9   g_signal_connect (app, "shutdown", G_CALLBACK (app_shutdown), NULL);
10  stat =g_application_run (G_APPLICATION (app), argc, argv);
11  g_object_unref (app);
12  return stat;
13 }
```

It connects three signals and handlers.

- startup: It is emitted after the application is registered to the system.
- activate: It is emitted when the application is activated.
- shutdown: It is emitted just before the application quits.

```
1  static void
2  app_startup (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkBuilder *build;
5    GtkWindow *win;
6    GtkDrawingArea *da;
7    GtkGesture *drag;
8
9    build = gtk_builder_new_from_resource ("/com/github/ToshioCP/rect/rect.ui");
10   win = GTK_WINDOW (gtk_builder_get_object (build, "win"));
11   da = GTK_DRAWING_AREA (gtk_builder_get_object (build, "da"));
12   gtk_window_set_application (win, app);
13   g_object_unref (build);
14
15   gtk_drawing_area_set_draw_func (da, draw_cb, NULL, NULL);
16   g_signal_connect_after (da, "resize", G_CALLBACK (resize_cb), NULL);
17
18   drag = gtk_gesture_drag_new ();
19   gtk_gesture_single_set_button (GTK_GESTURE_SINGLE (drag), GDK_BUTTON_PRIMARY);
20   gtk_widget_add_controller (GTK_WIDGET (da), GTK_EVENT_CONTROLLER (drag));
21   g_signal_connect (drag, "drag-begin", G_CALLBACK (drag_begin), NULL);
22   g_signal_connect (drag, "drag-update", G_CALLBACK (drag_update), da);
23   g_signal_connect (drag, "drag-end", G_CALLBACK (drag_end), da);
24 }
```

The startup handler does three things.

- Builds the widgets.
- Initializes the GtkDrawingArea instance.
  - Sets the drawing function
  - Connects the “resize” signal and the handler.
- Creates the GtkGestureDrag instance and initializes it. Gesture will be explained in this section later.

```
1  static void
2  app_activate (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkWindow *win;
5
6    win = gtk_application_get_active_window (app);
7    gtk_window_present (win);
8  }
```

The activate handler just shows the window.

26.3.2 GtkDrawingArea

The program has two cairo surfaces and they are pointed by the global variables.

```
static cairo_surface_t *surface = NULL;
static cairo_surface_t *surface_save = NULL;
```

The drawing process is as follows.

- Creates an image on `surface`.
- Copies `surface` to the cairo surface of the GtkDrawingArea.
- Calls `gtk_widget_queue_draw (da)` to draw it if necessary.

They are created in the "resize" signal handler.

```
1  static void
2  resize_cb (GtkWidget *widget, int width, int height, gpointer user_data) {
3    cairo_t *cr;
4
5    if (surface)
6      cairo_surface_destroy (surface);
7    surface = cairo_image_surface_create (CAIRO_FORMAT_RGB24, width, height);
8    if (surface_save)
9      cairo_surface_destroy (surface_save);
10   surface_save = cairo_image_surface_create (CAIRO_FORMAT_RGB24, width, height);
11   /* Paint the surface white. It is the background color. */
12   cr = cairo_create (surface);
13   cairo_set_source_rgb (cr, 1.0, 1.0, 1.0);
14   cairo_paint (cr);
15   cairo_destroy (cr);
16 }
```

This callback is called when the GtkDrawingArea is shown. It is the only call because the window is not resizable.

It creates image surfaces for `surface` and `surface_save`. The `surface` surface is painted white, which is the background color.

The drawing function copies `surface` to the GtkDrawingArea surface.

```
1  static void
2  draw_cb (GtkDrawingArea *da, cairo_t *cr, int width, int height, gpointer user_data)
       {
3    if (surface) {
4      cairo_set_source_surface (cr, surface, 0.0, 0.0);
5      cairo_paint (cr);
6    }
7  }
```

This function is called by the system when it needs to redraw the drawing area.

Two surfaces `surface` and `surface_save` are destroyed before the application quits.

```
1  static void
2  app_shutdown (GApplication *application) {
3    if (surface)
4      cairo_surface_destroy (surface);
5    if (surface_save)
6      cairo_surface_destroy (surface_save);
7  }
```

26.3.3 GtkGestureDrag

Gesture class is used to recognize human gestures such as click, drag, pan, swipe and so on. It is a subclass of GtkEventController. GtkGesture class is abstract and there are several implementations.

- GtkGestureClick
- GtkGestureDrag
- GtkGesturePan
- GtkGestureSwipe

- other implementations

The program `rect.c` uses GtkGestureDrag. It is the implementation for drags. The parent-child relationship is as follows.

```
GObject -- GtkEventController -- GtkGesture -- GtkGestureSingle -- GtkGestureDrag
```

GtkGestureSingle is a subclass of GtkGesture and optimized for singe-touch and mouse gestures.

A GtkGestureDrag instance is created and initialized in the startup signal handler in `rect.c`. See line 18 to 23 in the following.

```
1  static void
2  app_startup (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkBuilder *build;
5    GtkWindow *win;
6    GtkDrawingArea *da;
7    GtkGesture *drag;
8
9    build = gtk_builder_new_from_resource ("/com/github/ToshioCP/rect/rect.ui");
10   win = GTK_WINDOW (gtk_builder_get_object (build, "win"));
11   da = GTK_DRAWING_AREA (gtk_builder_get_object (build, "da"));
12   gtk_window_set_application (win, app);
13   g_object_unref (build);
14
15   gtk_drawing_area_set_draw_func (da, draw_cb, NULL, NULL);
16   g_signal_connect_after (da, "resize", G_CALLBACK (resize_cb), NULL);
17
18   drag = gtk_gesture_drag_new ();
19   gtk_gesture_single_set_button (GTK_GESTURE_SINGLE (drag), GDK_BUTTON_PRIMARY);
20   gtk_widget_add_controller (GTK_WIDGET (da), GTK_EVENT_CONTROLLER (drag));
21   g_signal_connect (drag, "drag-begin", G_CALLBACK (drag_begin), NULL);
22   g_signal_connect (drag, "drag-update", G_CALLBACK (drag_update), da);
23   g_signal_connect (drag, "drag-end", G_CALLBACK (drag_end), da);
24 }
```

- The function `gtk_gesture_drag_new` creates a new GtkGestureDrag instance.
- The function `gtk_gesture_single_set_button` sets the button number to listen to. The constant `GDK_BUTTON_PRIMARY` is the left button of a mouse.
- The function `gtk_widget_add_controller` adds an event controller, gestures are descendants of the event controller, to a widget.
- Three signals and handlers are connected.
  - drag-begin: Emitted when dragging starts.
  - drag-update: Emitted when the dragging point moves.
  - drag-end: Emitted when the dragging ends.

The process during the drag is as follows.

- start: save the surface and start points
- update: restore the surface and draw a thin rectangle between the start point and the current point of the mouse
- end: restore the surface and draw a thick rectangle between the start and end points.

We need two global variables for the start point.

```
static double start_x;
static double start_y;
```

The following is the handler for the “drag-begin” signal.

```
1  static void
2  copy_surface (cairo_surface_t *src, cairo_surface_t *dst) {
3    if (!src || !dst)
4      return;
5    cairo_t *cr = cairo_create (dst);
```

```
6     cairo_set_source_surface (cr, src, 0.0, 0.0);
7     cairo_paint (cr);
8     cairo_destroy (cr);
9   }
10
11  static void
12  drag_begin (GtkGestureDrag *gesture, double x, double y, gpointer user_data) {
13    // save the surface and record (x, y)
14    copy_surface (surface, surface_save);
15    start_x = x;
16    start_y = y;
17  }
```

- Copies `surface` to `surface_save`, which is an image just before the dragging.
- Stores the points to `start_x` and `start_y`.

```
1   static void
2   drag_update  (GtkGestureDrag *gesture, double offset_x, double offset_y, gpointer
        user_data) {
3     GtkWidget *da = GTK_WIDGET (user_data);
4     cairo_t *cr;
5
6     copy_surface (surface_save, surface);
7     cr = cairo_create (surface);
8     cairo_rectangle (cr, start_x, start_y, offset_x, offset_y);
9     cairo_set_line_width (cr, 1.0);
10    cairo_stroke (cr);
11    cairo_destroy (cr);
12    gtk_widget_queue_draw (da);
13  }
```

- Restores `surface` from `surface_save`.
- Draws a rectangle with thin lines.
- Calls `gtk_widget_queue_draw` to add the GtkDrawingArea to the queue to redraw.

```
1   static void
2   drag_end  (GtkGestureDrag *gesture, double offset_x, double offset_y, gpointer
        user_data) {
3     GtkWidget *da = GTK_WIDGET (user_data);
4     cairo_t *cr;
5
6     copy_surface (surface_save, surface);
7     cr = cairo_create (surface);
8     cairo_rectangle (cr, start_x, start_y, offset_x, offset_y);
9     cairo_set_line_width (cr, 6.0);
10    cairo_stroke (cr);
11    cairo_destroy (cr);
12    gtk_widget_queue_draw (da);
13  }
```

- Restores `surface` from `surface_save`.
- Draws a rectangle with thick lines.
- Calls `gtk_widget_queue_draw` to add the GtkDrawingArea to the queue to redraw.

26.4 Build and run

Download the repository. Change your current directory to `src/custom_drawing`. Run meson and ninja to build the program. Type `_build/rect` to run the program. Try to draw rectangles.

```
$ cd src/custom_drawing
$ meson setup _build
$ ninja -C _build
$ _build/rect
```

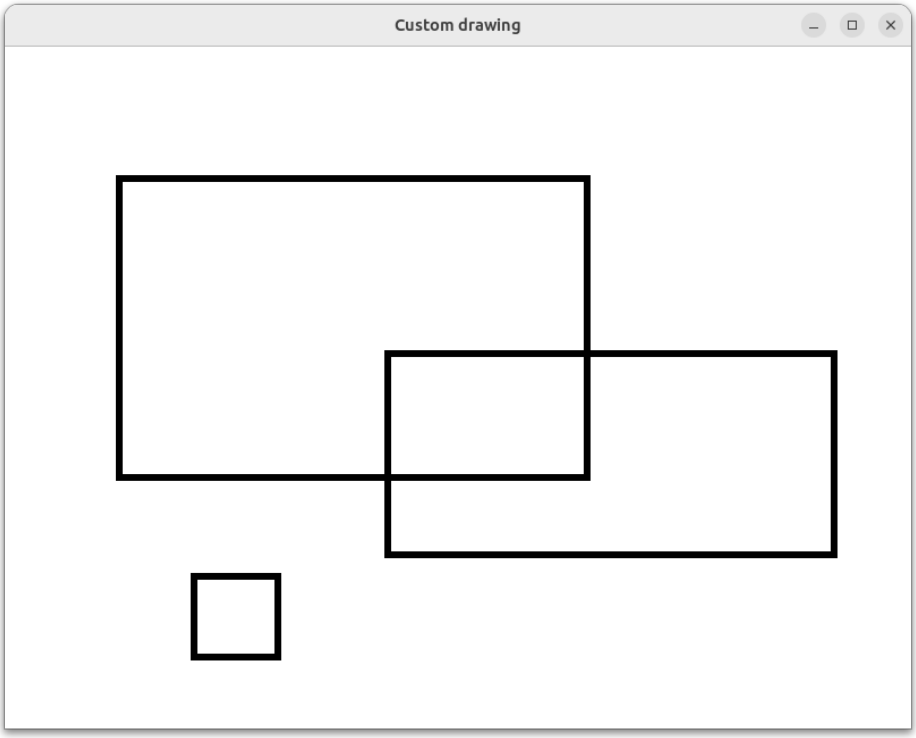


Figure 38: The screen of rect program

27 Tiny turtle graphics interpreter

A program `turtle` is an example with the combination of TfeTextView and GtkDrawingArea objects. It is a very small interpreter but you can draw fractal curves with it. The following diagram is a Koch curve, which is one of the famous fractal curves.

The following is a snow-crystal-shaped curve. It is composed of six Koch curves.

This program uses flex and bison. Flex is a lexical analyzer. Bison is a parser generator. These two programs are similar to lex and yacc which are proprietary software developed in Bell Laboratory. However, flex and bison are open source software. This section describes them and they are not the topics about GTK 4. So, readers can skip this section.

27.1 How to use turtle

The turtle document is in the appendix. I'll show you a simple example.

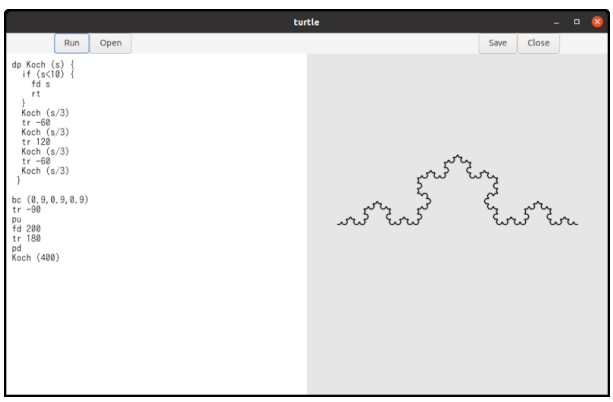


Figure 39: Koch curve

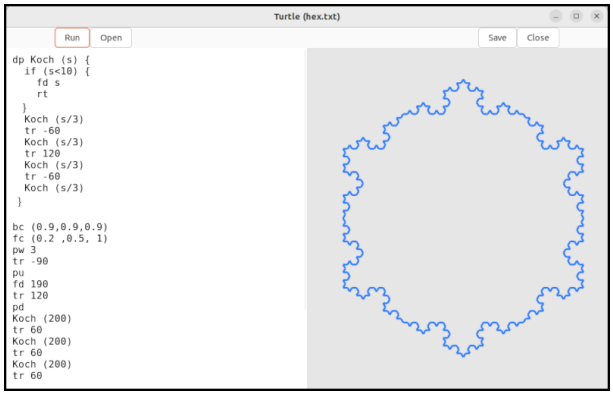


Figure 40: Snow

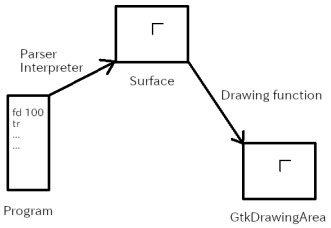


Figure 41: Parser, interpreter and drawing function

```
fc (1,0,0) # Foreground color is red, rgb = (1,0,0).
pd         # Pen down.
rp (4) {   # Repeat four times.
  fd 100   # Go forward by 100 pixels.
  tr 90    # Turn right by 90 degrees.
}
```

1. Compile and install `turtle` (See the documentation above). Then, run `turtle`.
2. Type the program above in the editor (left part of the window).
3. Click on the `Run` button, then a red square appears on the right part of the window. The side of the square is 100 pixels long.

In the same way, you can draw other curves. The turtle document includes some fractal curves such as tree, snow and square-koch. The source codes are located at src/turtle/example directory. You can read these files into `turtle` editor by clicking on the `Open` button.

27.2 Combination of TfeTextView and GtkDrawingArea objects

Turtle uses TfeTextView and GtkDrawingArea.

1. A user inputs/reads a turtle program into the buffer in the TfeTextView instance.
2. The user clicks on the “Run” button.
3. The parser reads the program and generates tree-structured data.
4. The interpriter reads the data and executes it step by step. And it draws shapes on a surface. The surface isn’t the one in the GtkDrawingArea widget.
5. The widget is added to the queue. It will be redrawn with the drawing function, which just copies the surface into the one in the GtkDrawingArea.

The body of the interpreter is written with flex and bison. The codes are not thread safe. So the callback function `run_cb`, which is the handler of “clicked” signal on the `Run` button, prevents reentering.

```
1 void
2 run_cb (GtkWidget *btnr) {
3   GtkTextBuffer *tb = gtk_text_view_get_buffer (GTK_TEXT_VIEW (tv));
4   GtkTextIter start_iter;
```

```
5    GtkTextIter end_iter;
6    char *contents;
7    int stat;
8    static gboolean busy = FALSE; /* initialized only once */
9    cairo_t *cr;
10
11   /* yyparse() and run() are NOT thread safe. */
12   /* The variable busy avoids reentrance. */
13   if (busy)
14     return;
15   busy = TRUE;
16   gtk_text_buffer_get_bounds (tb, &start_iter, &end_iter);
17   contents = gtk_text_buffer_get_text (tb, &start_iter, &end_iter, FALSE);
18   if (surface && contents[0] != '\0') {
19     init_flex (contents);
20     stat = yyparse ();
21     if (stat == 0) { /* No error */
22       run ();
23     }
24     finalize_flex ();
25   } else if (surface) {
26     cr = cairo_create (surface);
27     cairo_set_source_rgb (cr, 1.0, 1.0, 1.0);
28     cairo_paint (cr);
29     cairo_destroy (cr);
30   }
31   g_free (contents);
32   gtk_widget_queue_draw (GTK_WIDGET (da));
33   busy = FALSE;
34 }
35
36 static void
37 resize_cb (GtkDrawingArea *drawing_area, int width, int height, gpointer user_data) {
38
39   if (surface)
40     cairo_surface_destroy (surface);
41   surface = cairo_image_surface_create (CAIRO_FORMAT_ARGB32, width, height);
42   run_cb (NULL); // NULL is a fake (run button).
43 }
```

- 8, 13-15: The static value `busy` holds a status of the interpreter. If it is `TRUE`, the interpreter is running and it is not possible to call the interpreter because it's not a re-entrant program. If it is `FALSE`, it is safe to call the interpreter and set the variable `busy` to TRUE.
- 16-17: Gets the contents of `tb`.
- 18-30: The variable `surface` is a static variable. It points to a `cairo_surface_t` instance. It is created when the GtkDrawingArea instance is realized and whenever it is resized. Therefore, `surface` isn't NULL usually. But if it is NULL, the interpreter won't be called.
- 18-24: If `surface` points a surface instance and the string `contents` isn't empty, it calls the interpreter.
  - Initializes the lexical analyzer.
  - Calls the parser. The parser analyzes the program codes syntactically and generates a tree structured data.
  - If the parser successfully parsed, it calls the runtime routine 'run'.
  - Finalizes the lexical analyzer.
- 25-29: If `surface` points a surface instance and the string `contents` is empty, it clears the surface `surface`.
- 31: Frees `contents`.
- 32: Adds the drawing area widget to the queue to draw.
- 33: Sets the variable `busy` to FALSE.
- 36-43: The "resized" signal handler. If the `surface` isn't NULL, it is destroyed. A new surface is created. Its size is the same as the surface of the GtkDrawingArea instance. It calls the callback function `run_cb` to redraw the shape on the drawing area.

If the open button is clicked and a file is read, the filename will be shown on the header bar.

```
1  static void
2  show_filename (TfeTextView *tv) {
3    GFile *file;
4    char *filename;
5    char *title;
6
7    file = tfe_text_view_get_file (tv);
8    if (G_IS_FILE (file)) {
9      filename = g_file_get_basename (file);
10     title = g_strdup_printf ("Turtle␣(%s)", filename);
11     g_free (filename);
12     g_object_unref (file);
13   } else
14     title = g_strdup ("Turtle");
15   gtk_window_set_title (GTK_WINDOW (win), title);
16   g_free (title);
17 }
```

This function is the callback function of the "change-file" signal on the TfeTextView instance. It calls `tfe_text_view_get_file`.

- If the return value is a GFile instance, the title will be "Turtle (the filename)".
- Otherwise, the title will be "Turtle".

Other part of `turtleapplication.c` is very simple and similar to the codes in the former applications. The codes of `turtleapplication.c` is in the turtle directory.

27.3 What does the interpreter do?

Suppose that the turtle application runs with the following program.

```
distance = 100
fd distance*2
```

The application recognizes the program and works as follows.

- Generally, a program consists of tokens. Tokens are "distance", "=", "100", "fd", "*" and "2" in the above example..
- The parser calls a function `yylex` to read a token in the source file. `yylex` returns a code which is called "token kind" and sets a global variable `yylval` to a value, which is called a semantic value. The type of `yylval` is union. The type of `yylval.ID` and `yylval.NUM` are string and double respectively. There are seven tokens in the program so `yylex` is called seven times.

	token kind	yylval.ID	yylval.NUM
1	ID	distance	
2	=		
3	NUM		100
4	FD		
5	ID	distance	
6	*		
7	NUM		2

- The function `yylex` returns a token kind every time, but it doesn't set `yylval.ID` or `yylval.NUM` every time. It is because keywords (`FD`) and symbols (`=` and `*`) don't have any semantic values. The function `yylex` is called lexical analyzer or scanner.
- The application `turtle` makes a tree structured data. This part of `turtle` is called parser.
- `Turtle` analyzes the tree and executes it. This part of `turtle` is called runtime routine or interpreter. The tree consists of rectangles and line segments between the rectangles. The rectangles are called nodes. For example, N_PROGRAM, N_ASSIGN, N_FD and N_MUL are nodes.
  1. Goes down from N_PROGRAM to N_ASSIGN.

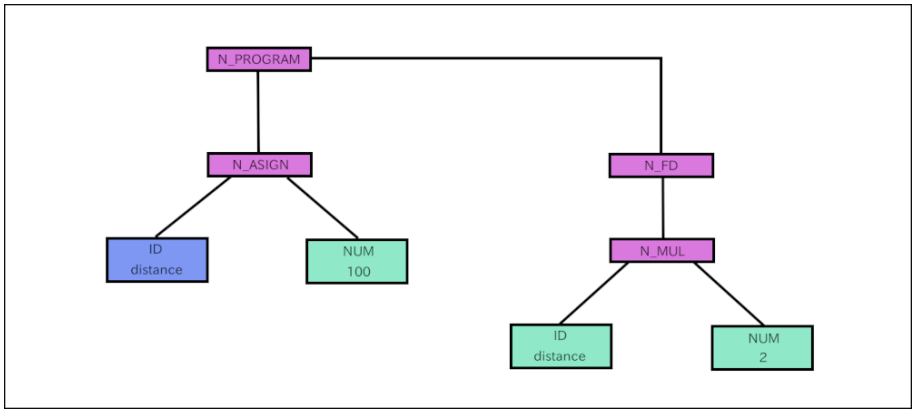


Figure 42: turtle parser tree

2. N_ASSIGN node has two children, ID and NUM. This node comes from "distance = 100" which is "ID = NUM" syntactically. First, `turtle` checks if the first child is ID. If it's ID, then `turtle` looks for the variable in the variable table. If it doesn't exist, it registers the ID (`distance`) to the table. Then go back to the N_ASSIGN node.
3. `Turtle` calculates the second child. In this case its a number 100. Saves 100 to the variable table at the `distance` record.
4. `Turtle` goes back to N_PROGRAM then go to the next node N_FD. It has only one child. Goes down to the child N_MUL.
5. The first child is ID (distance). Searches the variable table for the variable `distance` and gets the value 100. The second child is a number 2. Multiplies 100 by 2 and gets 200. Then `turtle` goes back to N_FD.
6. Now `turtle` knows the distance is 200. It moves the cursor forward by 200 pixels. The segment is drawn on the `surface`.
7. There are no node follows. Runtime routine returns to the function `run_cb`.

- The function `run_cb` calls `gtk_widget_queue_draw` and put the GtkDrawingArea widget to the queue.
- The system redraws the widget. At that time drawing function `draw_func` is called. The function copies the `surface` to the surface in the GtkDrawingArea.

Actual turtle program is more complicated than the example above. However, what turtle does is basically the same. Interpretation consists of three parts.

- Lexical analysis
- Syntax Parsing and tree generation
- Interpretation and execution of the tree.

27.4 Compilation flow

The source files are:

- flex source file => `turtle.lex`
- bison source file => `turtle.y`
- C header file => `turtle_lex.h`
- C source file => `turtleapplication.c`
- other files => `turtle.ui`, `turtle.gresources.xml` and `meson.build`

The compilation process is a bit complicated.

1. glib-compile-resources compiles `turtle.ui` to `resources.c` according to `turtle.gresource.xml`. It also generates `resources.h`.
2. bison compiles `turtle.y` to `turtle_parser.c` and generates `turtle_parser.h`
3. flex compiles `turtle.lex` to `turtle_lex.c`.
4. gcc compiles `application.c`, `resources.c`, `turtle_parser.c` and `turtle_lex.c` with `turtle_lex.h`, `resources.h` and `turtle_parser.h`. It generates an executable file `turtle`.

Meson controls the process. The instruction is described in `meson.build`.

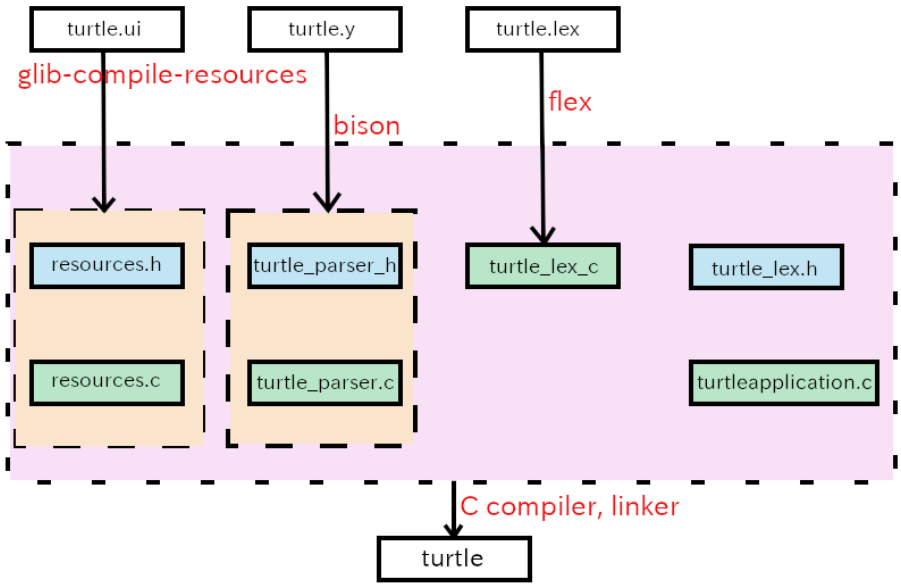


Figure 43: compile process

```
1  project('turtle', 'c')
2
3  compiler = meson.get_compiler('c')
4  mathdep = compiler.find_library('m', required : true)
5
6  gtkdep = dependency('gtk4')
7
8  gnome=import('gnome')
9  resources = gnome.compile_resources('resources','turtle.gresource.xml')
10
11 flex = find_program('flex')
12 bison = find_program('bison')
13 turtleparser = custom_target('turtleparser', input: 'turtle.y', output:
       ['turtle_parser.c', 'turtle_parser.h'], command: [bison, '-d', '-o',
       'turtle_parser.c', '@INPUT@'])
14 turtlelexer = custom_target('turtlelexer', input: 'turtle.lex', output:
       'turtle_lex.c', command: [flex, '-o', '@OUTPUT@', '@INPUT@'])
15
16 sourcefiles=files('turtleapplication.c', '../tfetextview/tfetextview.c')
17
18 executable('turtle', sourcefiles, resources, turtleparser, turtlelexer,
       turtleparser[1], dependencies: [mathdep, gtkdep], export_dynamic: true, install:
       true)
```

- 1: The project name is “turtle” and the program language is C.
- 3: Gets C compiler. It is usually `gcc` in linux.
- 4: Gets math library. This program uses trigonometric functions. They are defined in the math library, but the library is optional. So, it is necessary to include it by `#include <math.h>` and also link the library with the linker.
- 6: Gets gtk4 library.
- 8: Gets gnome module.See Meson build system website – GNOME module for further information.
- 9: Compiles ui file to C source file according to the XML file `turtle.gresource.xml`.
- 11: Gets flex.
- 12: Gets bison.
- 13: Compiles `turtle.y` to `turtle_parser.c` and `turtle_parser.h` by bison. The function `custom_target` creates a custom top level target. See Meson build system website – custom target for further

information.

- 14: Compiles `turtle.lex` to `turtle_lex.c` by flex.
- 16: The variable `sourcefiles` is a file object created with the C source files.
- 18: Compiles C source files including generated files by glib-compile-resources, bison and flex. The argument `turtleparser[1]` refers to `tirtle_parser.h` which is the second output in the line 13.

27.5 Turtle.lex

27.5.1 What does flex do?

Flex creates lexical analyzer from flex source file. Flex source file is a text file. Its syntactic rule will be explained later. Generated lexical analyzer is a C source file. It is also called scanner. It reads a text file, which is a source file of a program language, and gets variable names, numbers and symbols. Suppose here is a turtle source file.

```
fc (1,0,0) # Foreground color is red, rgb = (1,0,0).
pd         # Pen down.
distance = 100
angle = 90
fd distance    # Go forward by distance (100) pixels.
tr angle     # Turn right by angle (90) degrees.
```

The content of the text file is separated into `fc`, `(`, `1` and so on. The words `fc`, `pd`, `distance`, `angle`, `tr`, `1`, `0`, `100` and `90` are called tokens. The characters '`(`' (left parenthesis), '`,`' (comma), '`)`' (right parenthesis) and '`=`' (equal sign) are called symbols. (Sometimes those symbols called tokens, too.)

Flex reads `turtle.lex` and generates the C source file of a scanner. The file `turtle.lex` specifies tokens, symbols and the behavior which corresponds to each token or symbol. Turtle.lex isn't a big program.

```
1  %top{
2  #include <string.h>
3  #include <stdlib.h>
4  #include <glib.h>
5  #include "turtle_parser.h"
6
7    static int nline = 1;
8    static int ncolumn = 1;
9    static void get_location (char *text);
10
11   /* Dinamically allocated memories are added to the single list. They will be freed
         in the finalize function. */
12   extern GSList *list;
13 }
14
15 %option noyywrap
16
17 REAL_NUMBER (0|[1-9][0-9]*)(\.[0-9]+)?
18 IDENTIFIER [a-zA-Z][a-zA-Z0-9]*
19 %%
20   /* rules */
21 #.*               ; /* comment. Be careful. Dot symbol (.) matches any character but
       new line. */
22 [ ]               ncolumn++; /* white space. [ and ] is a "character class". */
23 \t                ncolumn += 8; /* assume that tab is 8 spaces. */
24 \n                nline++; ncolumn = 1;
25   /* reserved keywords */
26 pu                get_location (yytext); return PU; /* pen up */
27 pd                get_location (yytext); return PD; /* pen down */
28 pw                get_location (yytext); return PW; /* pen width = line width */
29 fd                get_location (yytext); return FD; /* forward */
30 tr                get_location (yytext); return TR; /* turn right */
31 tl                get_location (yytext); return TL; /* turn left, since ver 0.5 */
32 bc                get_location (yytext); return BC; /* background color */
33 fc                get_location (yytext); return FC; /* foreground color */
34 dp                get_location (yytext); return DP; /* define procedure */
```

```
35 if                get_location (yytext); return IF; /* if statement */
36 rt                get_location (yytext); return RT; /* return statement */
37 rs                get_location (yytext); return RS; /* reset the status */
38 rp                get_location (yytext); return RP; /* repeat, since ver 0.5 */
39   /* constant */
40 {REAL_NUMBER}     get_location (yytext); yylval.NUM = atof (yytext); return NUM;
41   /* identifier */
42 {IDENTIFIER}      { get_location (yytext); yylval.ID = g_strdup(yytext);
43                     list = g_slist_prepend (list, yylval.ID);
44                     return ID;
45                   }
46 "="               get_location (yytext); return '=';
47 ">"               get_location (yytext); return '>';
48 "<"               get_location (yytext); return '<';
49 "+"               get_location (yytext); return '+';
50 "-"               get_location (yytext); return '-';
51 "*"               get_location (yytext); return '*';
52 "/"               get_location (yytext); return '/';
53 "("               get_location (yytext); return '(';
54 ")"               get_location (yytext); return ')';
55 "{"               get_location (yytext); return '{';
56 "}"               get_location (yytext); return '}';
57 ","               get_location (yytext); return ',';
58 .                 ncolumn++;             return YYUNDEF;
59 %%
60
61 static void
62 get_location (char *text) {
63   yylloc.first_line = yylloc.last_line = nline;
64   yylloc.first_column = ncolumn;
65   yylloc.last_column = (ncolumn += strlen(text)) - 1;
66 }
67
68 static YY_BUFFER_STATE state;
69
70 void
71 init_flex (const char *text) {
72   state = yy_scan_string (text);
73 }
74
75 void
76 finalize_flex (void) {
77   yy_delete_buffer (state);
78 }
```

The file consists of three sections which are separated by "%%" (line 19 and 59). They are definitions, rules and user code sections.

27.5.2 Definitions section

- 1-12: Lines between "%top{" and "}" are C source codes. They will be copied to the top of the generated C source file.
- 2-3: This program uses two functions `strlen` (l.65) and `atof` (l.40). They are defined in `string.h` and `stdlib.h` respectively. These two header files are included here.
- 4: This program uses some GLib functions and structures like `g_strdup` and `GSList`. GLib header file is `glib.h` and it is included here.
- 5: The header file "turtle_parser.h" is generated from "turtle.y" by bison. It defines some constants and functions like `PU` and `yylloc`. The header file is included here.
- 7-9: The current input position is pointed by `nline` and `ncolumn`. The function `get_location` is declared here so that it can be called before the function is defined (l.61-65).
- 12: GSlist is a structure for a singly-linked list. The variable `list` is defined in `turtle.y` so its class is `extern`. It is the start point of the list. The list is used to keep allocated memories.
- 15: This option `%option noyywrap` must be specified when you have only single source file to the

scanner. Refer to "9 The Generated Scanner" in the flex documentation in your distribution. (The documentation is not on the internet.)

- 17-18: `REAL_NUMBER` and `IDENTIFIER` are names. A name begins with a letter or an underscore followed by zero or more letters, digits, underscores (_) or dashes (-). They are followed by regular expressions which are their definitions. They will be used in rules section and will expand to the definition.

27.5.3 Rules section

This section is the most important part. Rules consist of patterns and actions. The patterns are regular expressions or names surrounded by braces. The names must be defined in the definitions section. The definition of the regular expression is written in the flex documentation.

For example, line 40 is a rule.

- `{REAL_NUMBER}` is a pattern
- `get_location (yytext); yylval.NUM = atof (yytext); return NUM;` is an action.

`{REAL_NUMBER}` is defined in the line 17, so it expands to `(0|[1-9][0-9]*)(\.[0-9]+)?`. This regular expression matches numbers like `0`, `12` and `1.5`. If an input is a number, it matches the pattern in line 40. Then the matched text is assigned to `yytext` and corresponding action is executed. A function `get_location` changes the location variables to the position at the text. It assigns `atof (yytext)`, which is double sized number converted from `yytext`, to `yylval.NUM` and return `NUM`. `NUM` is a token kind and it represents (double type) numbers. It is defined in `turtle.y`.

The scanner generated by flex has `yylex` function. If `yylex` is called and the input is "123.4", then it works as follows.

1. A string "123.4" matches `{REAL_NUMBER}`.
2. Updates the location variable `ncolumn`. The structure `yylloc` is set by `get_location`.
3. The function `atof` converts the string "123.4" to double type number 123.4.
4. It is assigned to `yylval.NUM`.
5. `yylex` returns `NUM` to the caller.

Then the caller knows the input is a number (`NUM`), and its value is 123.4.

- 20-58: Rules section.
- 21: The symbol `.` (dot) matches any character except newline. Therefore, a comment begins `#` followed by any characters except newline. No action happens. That means that comments are ignored.
- 22: White space just increases the variable `ncolumn` by one.
- 23: Tab is assumed to be equal to eight spaces.
- 24: New line increases a variable `nline` by one and resets `ncolumn`.
- 26-38: Keywords updates the location variables `ncolumn` and `yylloc`, and returns the token kinds of the keywords.
- 40: Real number constant. The action converts the text`yytext` to a double type number, puts it into `yylval.NUM` and returns `NUM`.
- 42: `IDENTIFIER` is defined in line 18. The identifier is a name of variable or procedure. It begins with a letter and followed by letters or digits. The location variables are updated and the name of the identifier is assigned to `yylval.ID`. The memory of the name is allocated by the function `g_strdup`. The memory is registered to the list (GSlist type list). The memory will be freed after the runtime routine finishes. A token kind `ID` is returned.
- 46-57: Symbols update the location variable and return the token kinds. The token kind is the same as the symbol itself.
- 58: If the input doesn't match the patterns, then it is an error. A special token kind `YYUNDEF` is returned.

27.5.4 User code section

This section is just copied to C source file.

- 61-66: A function `get_location`. The location of the input is recorded to `nline` and `ncolumn`. A variable `yylloc` is referred by the parser. It is a C structure and has four members, `first_line`, `first_column`, `last_line` and `last_column`. They point the start and end of the current input text.

- 68: `YY_BUFFER_STATE` is a pointer points the input buffer. Flex makes the definition of `YY_BUFFER_STATE` in the C file (scanner source file `turtle_lex.c`). See your flex document, section 11 Multiple Input Buffers, for further information.
- 70-73: A function `init_flex` is called by `run_cb` which is a "clicked" signal handler on the `Run` button. It has one string type parameter. The caller assigns it with the content of the GtkTextBuffer instance. A function `yy_scan_string` sets the input buffer for the scanner.
- 75-78: A function `finalize_flex` is called after runtime routine finishes. It deletes the input buffer.

27.6 Turtle.y

Turtle.y has more than 800 lines so it is difficult to explain all the source code. So I will explain the key points and leave out other less important parts.

27.6.1 What does bison do?

Bison creates C source file of a parser from a bison source file. The bison source file is a text file. A parser analyzes a program source code according to its grammar. Suppose here is a turtle source file.

```
fc (1,0,0) # Foreground color is red, rgb = (1,0,0).
pd         # Pen down.
distance = 100
angle = 90
fd distance    # Go forward by distance (100) pixels.
tr angle     # Turn right by angle (90) degrees.
```

The parser calls `yylex` to get a token. The token consists of its type (token kind) and value (semantic value). So, the parser gets items in the following table whenever it calls `yylex`.

	token kind	yylval.ID	yylval.NUM
1	FC		
2	(		
3	NUM		1.0
4	,		
5	NUM		0.0
6	,		
7	NUM		0.0
8	)		
9	PD		
10	ID	distance	
11	=		
12	NUM		100.0
13	ID	angle	
14	=		
15	NUM		90.0
16	FD		
17	ID	distance	
18	TR		
19	ID	angle	

Bison source code specifies the grammar rules of turtle language. For example, `fc (1,0,0)` is called primary procedure. A procedure is like a void type C function. It doesn't return any values. Programmers can define their own procedures. On the other hand, `fc` is a built-in procedure. Such procedures are called primary procedures. It is described in bison source code like:

```
primary_procedure: FC '(' expression ',' expression ',' expression ')';
expression: ID | NUM;
```

This means:

- Primary procedure is FC followed by '(', expression, ',', expression, ',', expression and ')'.

- expression is ID or NUM.

The description above is called BNF (Backus-Naur form). Precisely speaking, it is not exactly the same as BNF. But the difference is small.

The first line is:

```
FC '(' NUM ',' NUM ',' NUM ')';
```

The parser analyzes the turtle source code and if the input matches the definition above, the parser recognizes it as a primary procedure.

The grammar of turtle is described in the Turtle manual. The following is an extract from the document.

```
program:
  statement
| program statement
;

statement:
  primary_procedure
| procedure_definition
;

primary_procedure:
  PU
| PD
| PW expression
| FD expression
| TR expression
| TL expression
| BC '(' expression ',' expression ',' expression ')'
| FC '(' expression ',' expression ',' expression ')'
| ID '=' expression
| IF '(' expression ')' '{' primary_procedure_list '}'
| RT
| RS
| RP '(' expression ')' '{' primary_procedure_list '}'
| ID '(' ')'
| ID '(' argument_list ')'
;

procedure_definition:
  DP ID '('  ')' '{' primary_procedure_list '}'
| DP ID '(' parameter_list ')' '{' primary_procedure_list '}'
;

parameter_list:
  ID
| parameter_list ',' ID
;

argument_list:
  expression
| argument_list ',' expression
;

primary_procedure_list:
  primary_procedure
| primary_procedure_list primary_procedure
;

expression:
  expression '=' expression
| expression '>' expression
| expression '<' expression
```

```
| expression '+' expression
| expression '-' expression
| expression '*' expression
| expression '/' expression
| '-' expression %prec UMINUS
| '(' expression ')'
| ID
| NUM
;
```

The grammar rule defines `program` first.

- program is a statement or a program followed by a statement.

The definition is recursive.

- `statement` is program.
- `statement statement` is `program statement`. Therefore, it is program.
- `statement statement statement` is `program statement` because the first two statements are `program`. Therefore, it is program.

You can find that a sequence of statements is program as well.

The symbols `program` and `statement` aren't tokens. They don't appear in the input. They are called non terminal symbols. On the other hand, tokens are called terminal symbols. The word "token" used here has wide meaning, it includes tokens and symbols which appear in the input. Non terminal symbols are often shortened to nterm.

Let's analyze the program above as bison does.

	token kind	yylval.ID	yylval.NUM	parse	S/R
1	FC			FC	S
2	(			FC(	S
3	NUM		1.0	FC(NUM	S
				FC(expression	R
4	,			FC(expression,	S
5	NUM		0.0	FC(expression,NUM	S
				FC(expression,expression	R
6	,			FC(expression,expression,	S
7	NUM		0.0	FC(expression,expression,NUM	S
				FC(expression,expression,expression	R
8	)			FC(expression,expression,expression)	S
				primary_procedure	R
				statement	R
				program	R
9	PD			program PD	S
				program primary_procedure	R
				program statement	R
				program	R
10	ID	distance		program ID	S
11	=			program ID=	S
12	NUM		100.0	program ID=NUM	S
				program ID=expression	R
				program primary_procedure	R
				program statement	R
				program	R
13	ID	angle		program ID	S
14	=			program ID=	S
15	NUM		90.0	program ID=NUM	S
				program ID=expression	R
				program primary_procedure	R
				program statement	R

	token kind	yylval.ID	yylval.NUM	parse	S/R
				program	R
16	FD			program FD	S
17	ID	distance		program FD ID	S
				program FD expression	R
				program primary_procedure	R
				program statement	R
				program	R
18	TR			program TR	S
19	ID	angle		program TR ID	S
				program TR expression	R
				program primary_procedure	R
				program statement	R
				program	R

The right most column shows shift/reduce. Shift is appending an input to the buffer. Reduce is substituting a higher nterm for the pattern in the buffer. For example, NUM is replaced by expression in the forth row. This substitution is "reduce".

Bison repeats shift and reduction until the end of the input. If the result is reduced to `program`, the input is syntactically valid. Bison executes an action whenever reduction occurs. Actions build a tree. The tree is analyzed and executed by runtime routine later.

Bison source files are called bison grammar files. A bison grammar file consists of four sections, prologue, declarations, rules and epilogue. The format is as follows.

```
%{
prologue
%}
declarations
%%
rules
%%
epilogue
```

27.6.2 Prologue

Prologue section consists of C codes and the codes are copied to the parser implementation file. You can use `%code` directives to qualify the prologue and identifies the purpose explicitly. The following is an extract from `turtle.y`.

```
%code top{
  #include <stdarg.h>
  #include <setjmp.h>
  #include <math.h>
  #include <glib.h>
  #include <cairo.h>
  #include "turtle_parser.h"

  /* The following line defines 'debug' so that debug information is printed out
      during the run time. */
  /* However it makes the program slow. */
  /* If you want to debug on, uncomment the line. */

  /* #define debug 1 */

  extern cairo_surface_t *surface;

  /* error reporting */
  static void yyerror (char const *s) { /* for syntax error */
```

```
    g_printerr ("%s from line %d, column %d to line %d, column %d\n",s,
        yylloc.first_line, yylloc.first_column, yylloc.last_line,
        yylloc.last_column);
  }
  /* Node type */
  enum {
    N_PU,
    N_PD,
    N_PW,
 ... ... ...
  };
}
```

The directive `%code top` copies its contents to the top of the parser implementation file. It usually includes `#include` directives, declarations of functions and definitions of constants. A function `yyerror` reports a syntax error and is called by the parser. Node type identifies a node in the tree.

Another directive `%code requires` copies its contents to both the parser implementation file and header file. The header file is read by the scanner C source file and other files.

```
%code requires {
  int yylex (void);
  int yyparse (void);
  void run (void);

  /* semantic value type */
  typedef struct _node_t node_t;
  struct _node_t {
    int type;
    union {
      struct {
        node_t *child1, *child2, *child3;
      } child;
      char *name;
      double value;
    } content;
  };
}
```

- `yylex` is shared by the parser implementation file and scanner file.
- `yyparse` and `run` is called by `run_cb` in `turtleapplication.c`.
- `node_t` is the type of the semantic value of nterms. The header file defines `YYSTYPE`, which is the semantic value type, with all the token and nterm value types. The following is extracted from the header file.

```
/* Value type.  */
#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
union YYSTYPE
{
  char * ID;                                /* ID  */
  double NUM;                               /* NUM  */
  node_t * program;                         /* program  */
  node_t * statement;                       /* statement  */
  node_t * primary_procedure;               /* primary_procedure  */
  node_t * primary_procedure_list;          /* primary_procedure_list  */
  node_t * procedure_definition;            /* procedure_definition  */
  node_t * parameter_list;                  /* parameter_list  */
  node_t * argument_list;                   /* argument_list  */
  node_t * expression;                      /* expression  */
};
```

Other useful macros and declarations are put into the `%code` directive.

```
%code {
/* The following macro is convenient to get the member of the node. */
```

```
  #define child1(n) (n)->content.child.child1
  #define child2(n) (n)->content.child.child2
  #define child3(n) (n)->content.child.child3
  #define name(n) (n)->content.name
  #define value(n) (n)->content.value

  /* start of nodes */
  static node_t *node_top = NULL;
  /* functions to generate trees */
  static node_t *tree1 (int type, node_t *child1, node_t *child2, node_t *child3);
  static node_t *tree2 (int type, double value);
  static node_t *tree3 (int type, char *name);
}
```

27.6.3 Bison declarations

Bison declarations defines terminal and non-terminal symbols. It also specifies some directives.

```
%locations
%define api.value.type union /* YYSTYPE, the type of semantic values, is union of
    following types */
 /* key words */
%token PU
%token PD
%token PW
%token FD
%token TR
%token TL /* ver 0.5 */
%token BC
%token FC
%token DP
%token IF
%token RT
%token RS
%token RP /* ver 0.5 */
 /* constant */
%token <double> NUM
 /* identifier */
%token <char *> ID
 /* non terminal symbol */
%nterm <node_t *> program
%nterm <node_t *> statement
%nterm <node_t *> primary_procedure
%nterm <node_t *> primary_procedure_list
%nterm <node_t *> procedure_definition
%nterm <node_t *> parameter_list
%nterm <node_t *> argument_list
%nterm <node_t *> expression
 /* logical relation symbol */
%left '=' '<' '>'
 /* arithmetic symbol */
%left '+' '-'
%left '*' '/'
%precedence UMINUS /* unary minus */
```

`%locations` directive inserts the location structure into the header file. It is like this.

```
typedef struct YYLTYPE YYLTYPE;
struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
};
```

This type is shared by the scanner file and the parser implementation file. The error report function `yyerror` uses it so that it can inform the location that error occurs.

`%define api.value.type union` generates semantic value type with tokens and nterms and inserts it to the header file. The inserted part is shown in the previous subsection as the extracts that shows the value type (YYSTYPE).

`%token` and `%nterm` directives define tokens and non terminal symbols respectively.

```
%token PU
... ...
%token <double> NUM
```

These directives define a token `PU` and `NUM`. The values of token kinds `PU` and `NUM` are defined as an enumeration constant in the header file.

```
  enum yytokentype
  {
  ... ... ...
    PU = 258,                      /* PU  */
  ... ... ...
    NUM = 269,                     /* NUM  */
  ... ... ...
  };
  typedef enum yytokentype yytoken_kind_t;
```

In addition, the type of the semantic value of `NUM` is defined as double in the header file because of `<double>` tag.

```
union YYSTYPE
{
  char * ID;                                /* ID  */
  double NUM;                               /* NUM  */
  ... ...
}
```

All the nterm symbols have the same type `*node_t` of the semantic value.

`%left` and `%precedence` directives define the precedence of operation symbols.

```
 /* logical relation symbol */
%left '=' '<' '>'
 /* arithmetic symbol */
%left '+' '-'
%left '*' '/'
%precedence UMINUS /* unary minus */
```

`%left` directive defines the following symbols as left-associated operators. If an operator `+` is left-associated, then

```
A + B + C = (A + B) + C
```

That is, the calculation is carried out the left operator first, then the right operator. If an operator `*` is right-associated, then:

```
A * B * C = A * (B * C)
```

The definition above decides the behavior of the parser. Addition and multiplication hold associative law so the result of `(A+B)+C` and `A+(B+C)` are equal in terms of mathematics. However, the parser will be confused if left (or right) associativity is not specified.

`%left` and `%precedence` directives show the precedence of operators. Later declared operators have higher precedence than former declared ones. The declaration above says, for example,

```
v=w+z*5+7 is the same as v=((w+(z*5))+7)
```

Be careful. The operator = above is an assignment. Assignment is not expression in turtle language. It is primary_procedure. But if = appears in an expression, it is a logical operator, not an assignment. The logical equal '=' usually used in the conditional expression, for example, in `if` statement. (Turtle language uses '=' instead of '==' in C language).

27.6.4 Grammar rules

Grammar rules section defines the syntactic grammar of the language. It is similar to BNF form.

```
result: components { action };
```

- result is a nterm.
- components are list of tokens or nterms.
- action is C codes. It is executed whenever the components are reduced to the result. Action can be left out.

The following is a part of the grammar rule in `turtle.y`. But it is not exactly the same.

```
program:
  statement { node_top = $$ = $1; }
;
statement:
  primary_procedure
;
primary_procedure:
  FD expression    { $$ = tree1 (N_FD, $2, NULL, NULL); }
;
expression:
  NUM   { $$ = tree2 (N_NUM, $1); }
;
```

- The first two lines tell that `program` is `statement`.
- Whenever `statement` is reduced to `program`, an action `node_top=$$=$1;` is executed.
- `node_top` is a static variable. It points the top node of the tree.
- The symbol `$$` is a semantic value of the result. For example, `$$` in line 2 is the semantic value of `program`. It is a pointer to a `node_t` type structure.
- The symbol `$1` is a semantic value of the first component. For example, `$1` in line 2 is the semantic value of `statement`. It is also a pointer to `node_t`.
- The next rule is that `statement` is `primary_procedure`. There's no action specified. Then, the default action `$$ = $1` is executed.
- The next rule is that `primary_procedure` is `FD` followed by expression. The action calls `tree1` and assigns its return value to `$$`. The function `tree1` makes a tree node. The tree node has type and union of three pointers to children nodes, string or double.

```
node --+-- type
       +-- union contents
                    +---struct {node_t *child1, *child2, *child3;};
                    +---char *name
                    +---double value
```

- `tree1` assigns the four arguments to type, child1, child2 and child3 members.
- The last rule is that `expression` is `NUM`.
- `tree2` makes a tree node. The paremeters of `tree2` are a type and a semantic value.

Suppose the parser reads the following program.

```
fd 100
```

What does the parser do?

1. The parser recognizes the input is `FD`. Maybe it is the start of `primary_procedure`, but parser needs to read the next token.
2. `yylex` returns the token kind `NUM` and sets `yylval.NUM` to 100.0 (the type is double). The parser reduces `NUM` to `expression`. At the same time, it sets the semantic value of the `expression` to point a new node. The node has the type `N_NUM` and a semantic value 100.0.

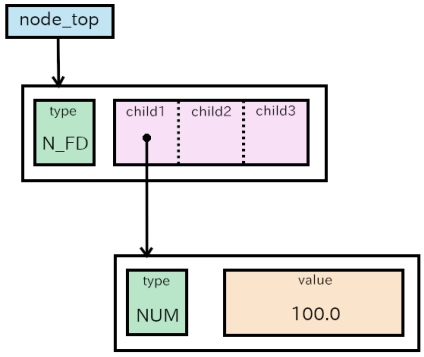


Figure 44: tree

3. After the reduction, the buffer has `FD` and `expression`. The parser reduces it to `primary_procedure`. And it sets the semantic value of the `primary_procedure` to point a new node. The node has the type `N_FD` and its member child1 points the node of `expression`, whose type is `N_NUM`.
4. The parser reduces `primary_procedure` to `statement`. The semantic value of `statement` is the same as the one of `primary_procedure`, which points to the node `N_FD`.
5. The parser reduces `statement` to `program`. The semantic value of `statement` is assigned to the one of `program` and the static variable `node_top`.
6. Finally `node_top` points the node `N_FD` and the node `N_FD` points the node `N_NUM`.

The following is the grammar rule extracted from `turtle.y`. The rules there are based on the same idea above. I don't want to explain the whole rules below. Please look into each line carefully so that you will understand all the rules and actions.

```
program:
  statement { node_top = $$ = $1; }
| program statement {
        node_top = $$ = tree1 (N_program, $1, $2, NULL);
#ifdef debug
if (node_top == NULL) g_printerr ("program: node_top is NULL.\n"); else g_printerr
    ("program: node_top is NOT NULL.\n");
#endif
        }
;

statement:
  primary_procedure
| procedure_definition
;

primary_procedure:
  PU    { $$ = tree1 (N_PU, NULL, NULL, NULL); }
| PD    { $$ = tree1 (N_PD, NULL, NULL, NULL); }
| PW expression    { $$ = tree1 (N_PW, $2, NULL, NULL); }
| FD expression    { $$ = tree1 (N_FD, $2, NULL, NULL); }
| TR expression    { $$ = tree1 (N_TR, $2, NULL, NULL); }
| TL expression    { $$ = tree1 (N_TL, $2, NULL, NULL); } /* ver 0.5 */
| BC '(' expression ',' expression ',' expression ')' { $$ = tree1 (N_BC, $3, $5,
    $7); }
| FC '(' expression ',' expression ',' expression ')' { $$ = tree1 (N_FC, $3, $5,
    $7); }
 /* assignment */
| ID '=' expression   { $$ = tree1 (N_ASSIGN, tree3 (N_ID, $1), $3, NULL); }
 /* control flow */
| IF '(' expression ')' '{' primary_procedure_list '}' { $$ = tree1 (N_IF, $3, $6,
    NULL); }
| RT    { $$ = tree1 (N_RT, NULL, NULL, NULL); }
```

```
| RS    { $$ = tree1 (N_RS, NULL, NULL, NULL); }
| RP '(' expression ')' '{' primary_procedure_list '}'    { $$ = tree1 (N_RP, $3,
    $6, NULL); }
 /* user defined procedure call */
| ID '(' ')'  { $$ = tree1 (N_procedure_call, tree3 (N_ID, $1), NULL, NULL); }
| ID '(' argument_list ')'  { $$ = tree1 (N_procedure_call, tree3 (N_ID, $1), $3,
    NULL); }
;

procedure_definition:
  DP ID '('  ')' '{' primary_procedure_list '}'  {
         $$ = tree1 (N_procedure_definition, tree3 (N_ID, $2), NULL, $6);
        }
| DP ID '(' parameter_list ')' '{' primary_procedure_list '}'  {
         $$ = tree1 (N_procedure_definition, tree3 (N_ID, $2), $4, $7);
        }
;

parameter_list:
  ID { $$ = tree3 (N_ID, $1); }
| parameter_list ',' ID  { $$ = tree1 (N_parameter_list, $1, tree3 (N_ID, $3),
    NULL); }
;

argument_list:
  expression
| argument_list ',' expression { $$ = tree1 (N_argument_list, $1, $3, NULL); }
;

primary_procedure_list:
  primary_procedure
| primary_procedure_list primary_procedure {
         $$ = tree1 (N_primary_procedure_list, $1, $2, NULL);
        }
;

expression:
  expression '=' expression { $$ = tree1 (N_EQ, $1, $3, NULL); }
| expression '>' expression { $$ = tree1 (N_GT, $1, $3, NULL); }
| expression '<' expression { $$ = tree1 (N_LT, $1, $3, NULL); }
| expression '+' expression { $$ = tree1 (N_ADD, $1, $3, NULL); }
| expression '-' expression { $$ = tree1 (N_SUB, $1, $3, NULL); }
| expression '*' expression { $$ = tree1 (N_MUL, $1, $3, NULL); }
| expression '/' expression { $$ = tree1 (N_DIV, $1, $3, NULL); }
| '-' expression %prec UMINUS { $$ = tree1 (N_UMINUS, $2, NULL, NULL); }
| '(' expression ')' { $$ = $2; }
| ID    { $$ = tree3 (N_ID, $1); }
| NUM   { $$ = tree2 (N_NUM, $1); }
;
```

27.6.5 Epilogue

The epilogue is written in C language and copied to the parser implementation file. Generally, you can put anything into the epilogue. In the case of turtle interpreter, the runtime routine and some other functions are in the epilogue.

Functions to create tree nodes There are three functions, `tree1`, `tree2` and `tree3`.

- `tree1` creates a node and sets the node type and pointers to its three children (NULL is possible).
- `tree2` creates a node and sets the node type and a value (double).
- `tree3` creates a node and sets the node type and a pointer to a string.

Each function gets memories first and build a node on them. The memories are inserted to the list. They will be freed when runtime routine finishes.

The three functions are called in the actions in the rules section.

```
/* Dynamically allocated memories are added to the single list. They will be freed
    in the finalize function. */
GSList *list = NULL;

node_t *
tree1 (int type, node_t *child1, node_t *child2, node_t *child3) {
  node_t *new_node;

  list = g_slist_prepend (list, g_malloc (sizeof (node_t)));
  new_node = (node_t *) list->data;
  new_node->type = type;
  child1(new_node) = child1;
  child2(new_node) = child2;
  child3(new_node) = child3;
  return new_node;
}

node_t *
tree2 (int type, double value) {
  node_t *new_node;

  list = g_slist_prepend (list, g_malloc (sizeof (node_t)));
  new_node = (node_t *) list->data;
  new_node->type = type;
  value(new_node) = value;
  return new_node;
}

node_t *
tree3 (int type, char *name) {
  node_t *new_node;

  list = g_slist_prepend (list, g_malloc (sizeof (node_t)));
  new_node = (node_t *) list->data;
  new_node->type = type;
  name(new_node) = name;
  return new_node;
}
```

Symbol table Variables and user defined procedures are registered in the symbol table. This table is a C array. It should be replaced by better algorithm and data structure, for example hash, in the future version

- Variables are registered with its name and value.
- Procedures are registered with its name and a pointer to the node of the procedure.

Therefore the table has the following fields.

- type to identify variable or procedure
- name
- value or pointer to a node

```
#define MAX_TABLE_SIZE 100
enum {
  PROC,
  VAR
};

struct {
  int type;
  char *name;
  union {
    node_t *node;
    double value;
```

```
  } object;
} table[MAX_TABLE_SIZE];
int tp;

void
init_table (void) {
  tp = 0;
}
```

The function `init_table` initializes the table. This must be called before registrations.

There are five functions to access the table,

- `proc_install` installs a procedure.
- `var_install` installs a variable.
- `proc_lookup` looks up a procedure. If the procedure is found, it returns a pointer to the node. Otherwise it returns NULL.
- `var_lookup` looks up a variable. If the variable is found, it returns TRUE and sets the pointer (argument) to point the value. Otherwise it returns FALSE.
- `var_replace` replaces the value of a variable. If the variable hasn't registered yet, it installs the variable.

```
int
tbl_lookup (int type, char *name) {
  int i;

  if (tp == 0)
    return -1;
  for (i=0; i<tp; ++i)
    if (type == table[i].type && strcmp(name, table[i].name) == 0)
      return i;
  return -1;
}

void
tbl_install (int type, char *name, node_t *node, double value) {
  if (tp >= MAX_TABLE_SIZE)
    runtime_error ("Symbol␣table␣overflow.\n");
  else if (tbl_lookup (type, name) >= 0)
    runtime_error ("Name␣%s␣is␣already␣registered.\n", name);
  else {
    table[tp].type = type;
    table[tp].name = name;
    if (type == PROC)
      table[tp++].object.node = node;
    else
      table[tp++].object.value = value;
  }
}

void
proc_install (char *name, node_t *node) {
  tbl_install (PROC, name, node, 0.0);
}

void
var_install (char *name, double value) {
  tbl_install (VAR, name, NULL, value);
}

void
var_replace (char *name, double value) {
  int i;
  if ((i = tbl_lookup (VAR, name)) >= 0)
    table[i].object.value = value;
```

```
  else
    var_install (name, value);
}

node_t *
proc_lookup (char *name) {
  int i;
  if ((i = tbl_lookup (PROC, name)) < 0)
    return NULL;
  else
    return table[i].object.node;
}

gboolean
var_lookup (char *name, double *value) {
  int i;
  if ((i = tbl_lookup (VAR, name)) < 0)
    return FALSE;
  else {
    *value = table[i].object.value;
    return TRUE;
  }
}
```

Stack for parameters and arguments Stack is a last-in first-out data structure. It is shortened to LIFO. Turtle uses a stack to keep parameters and arguments. They are like `auto` class variables in C language. They are pushed to the stack whenever the procedure is called. LIFO structure is useful for recursive calls.

Each element of the stack has name and value.

```
#define MAX_STACK_SIZE 500
struct {
  char *name;
  double value;
} stack[MAX_STACK_SIZE];
int sp, sp_biggest;

void
init_stack (void) {
  sp = sp_biggest = 0;
}
```

`sp` is a stack pointer. It is an index of the array `stack` and it always points an element of the array to store the next data. `sp_biggest` is the biggest number assigned to `sp`. We can know the amount of elements used in the array during the runtime. The purpose of the variable is to find appropriate `MAX_STACK_SIZE`. It will be unnecessary in the future version if the stack is implemented with better data structure and memory allocation.

The runtime routine push data to the stack when it executes a node of a procedure call. (The type of the node is `N_procedure_call`.)

```
dp drawline (angle, distance) { ... ... ... }
drawline (90, 100)
```

- The first line defines a procedure `drawline`. The runtime routine stores the name `drawline` and the node of the procedure to the symbol table.
- The second line calls the procedure. First, it looks for the procedure in the symbol table and gets its node. Then it searches the node for the parameters and gets `angle` and `distance`.
- It pushes ("angle", 90.0) to the stack.
- It pushes ("distance", 100.0) to the stack.
- It pushes (NULL, 2.0) to the stack. The number 2.0 is the number of parameters (or arguments). It is used when the procedure returns.

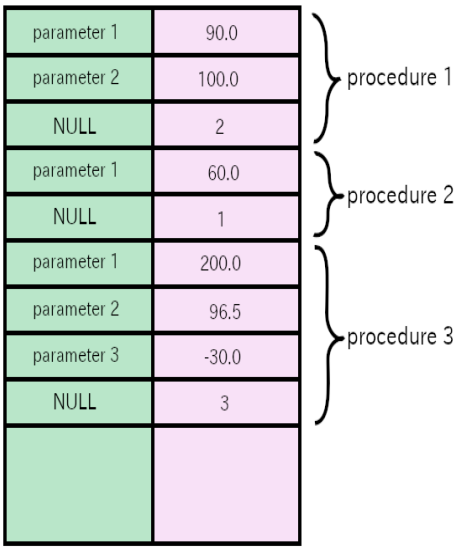


Figure 45: Stack

The following diagram shows the structure of the stack. First, `procedure 1` is called. The procedure has two parameters. In the `procedure 1`, another procedure `procedure 2` is called. It has one parameter. In the `procedure 2`, another procedure `procedure 3` is called. It has three parameters. These three procedures are nested.

Programs push data to a stack from a low address memory to a high address memory. In the following diagram, the lowest address is at the top and the highest address is at the bottom. That is the order of the address. However, "the top of the stack" is the last pushed data and "the bottom of the stack" is the first pushed data. Therefore, "the top of the stack" is the bottom of the rectangle in the diagram and "the bottom of the stack" is the top of the rectangle.

There are four functions to access the stack.

- `stack_push` pushes data to the stack.
- `stack_lookup` searches the stack for the variable given its name as an argument. It searches only the parameters of the latest procedure. It returns TRUE and sets the argument `value` to point the value, if the variable has been found. Otherwise it returns FALSE.
- `stack_replace` replaces the value of the variable in the stack. If it succeeds, it returns TRUE. Otherwise returns FALSE.
- `stack_return` throws away the latest parameters. The stack pointer goes back to the point before the latest procedure call so that it points to parameters of the previous called procedure.

```
void
stack_push (char *name, double value) {
  if (sp >= MAX_STACK_SIZE)
    runtime_error ("Stack␣overflow.\n");
  else {
    stack[sp].name = name;
    stack[sp++].value = value;
    sp_biggest = sp > sp_biggest ? sp : sp_biggest;
  }
}

int
stack_search (char *name) {
  int depth, i;

  if (sp == 0)
    return -1;
  depth = (int) stack[sp-1].value;
```

```
  if (depth + 1 > sp) /* something strange */
    runtime_error ("Stack error.\n");
  for (i=0; i<depth; ++i)
    if (strcmp(name, stack[sp-(i+2)].name) == 0) {
      return sp-(i+2);
    }
  return -1;
}

gboolean
stack_lookup (char *name, double *value) {
  int i;

  if ((i = stack_search (name)) < 0)
    return FALSE;
  else {
    *value = stack[i].value;
    return TRUE;
  }
}

gboolean
stack_replace (char *name, double value) {
  int i;

  if ((i = stack_search (name)) < 0)
    return FALSE;
  else {
    stack[i].value = value;
    return TRUE;
  }
}

void
stack_return(void) {
  int depth;

  if (sp <= 0)
    return;
  depth = (int) stack[sp-1].value;
  if (depth + 1 > sp) /* something strange */
    runtime_error ("Stack error.\n");
  sp -= depth + 1;
}
```

Surface and cairo A global variable `surface` is shared by `turtleapplication.c` and `turtle.y`. It is initialized in `turtleapplication.c`.

The runtime routine has its own cairo context. This is different from the cairo in the GtkDrawingArea instance. The runtime routine draws a shape on the `surface` with the cairo context. After runtime routine returns to `run_cb`, `run_cb` adds the GtkDrawingArea widget to the queue to redraw. When the widget is redraw,the drawing function `draw_func` is called. It copies the `surface` to the surface in the GtkDrawingArea object.

`turtle.y` has two functions `init_cairo` and `destroy_cairo`.

- `init_cairo` initializes static variables and cairo context. The variables keep pen status (up or down), direction, initial location, line width and color. The size of the `surface` changes according to the size of the window. Whenever a user drags and resizes the window, the `surface` is also resized. `init_cairo` gets the size first and sets the initial location of the turtle (center of the surface) and the transformation matrix.
- `destroy_cairo` just destroys the cairo context.

Turtle has its own coordinate. The origin is at the center of the surface, and positive direction of x and

$$\begin{bmatrix} z \\ w \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} p \\ q \end{bmatrix}$$

Figure 46: transformation

y axes are right and up respectively. But surfaces have its own coordinate. Its origin is at the top-left corner of the surface and positive direction of x and y are right and down respectively. A plane with the turtle's coordinate is called user space, which is the same as cairo's user space. A plane with the surface's coordinate is called device space.

Cairo provides a transformation which is an affine transformation. It transforms a user-space coordinate (x, y) into a device-space coordinate (z, w).

The function `init_cairo` gets the width and height of the `surface` (See the program below).

- The center of the surface is (0,0) with regard to the user-space coordinate and (width/2, height/2) with regard to the device-space coordinate.
- The positive direction of x axis in the two spaces are the same. So, (1,0) is transformed into (1+width/2,height/2).
- The positive direction of y axis in the two spaces are opposite. So, (0,1) is transformed into (width/2,-1+height/2).

You can determine a, b, c, d, p and q by substituting the numbers above for x, y, z and w in the equation above. The solution of the simultaneous equations is:

```
a = 1, b = 0, c = 0, d = -1, p = width/2, q = height/2
```

Cairo provides a structure `cairo_matrix_t`. The function `init_cairo` uses it and sets the cairo transformation (See the program below). Once the matrix is set, the transformation always performs whenever `cairo_stroke` function is invoked.

```
/* status of the surface */
static gboolean pen = TRUE;
static double angle = 90.0; /* angle starts from x axis and measured
    counterclockwise */
                  /* Initially facing to the north */
static double cur_x = 0.0;
static double cur_y = 0.0;
static double line_width = 2.0;

struct color {
  double red;
  double green;
  double blue;
};
static struct color bc = {0.95, 0.95, 0.95}; /* white */
static struct color fc = {0.0, 0.0, 0.0}; /* black */

/* cairo */
static cairo_t *cr;
gboolean
init_cairo (void) {
  int width, height;
  cairo_matrix_t matrix;

  pen = TRUE;
  angle = 90.0;
  cur_x = 0.0;
  cur_y = 0.0;
  line_width = 2.0;
  bc.red = 0.95; bc.green = 0.95; bc.blue = 0.95;
```

```
  fc.red = 0.0; fc.green = 0.0; fc.blue = 0.0;

  if (surface) {
    width = cairo_image_surface_get_width (surface);
    height = cairo_image_surface_get_height (surface);
    matrix.xx = 1.0; matrix.xy = 0.0; matrix.x0 = (double) width / 2.0;
    matrix.yx = 0.0; matrix.yy = -1.0; matrix.y0 = (double) height / 2.0;

    cr = cairo_create (surface);
    cairo_transform (cr, &matrix);
    cairo_set_source_rgb (cr, bc.red, bc.green, bc.blue);
    cairo_paint (cr);
    cairo_set_source_rgb (cr, fc.red, fc.green, fc.blue);
    cairo_move_to (cr, cur_x, cur_y);
    return TRUE;
  } else
    return FALSE;
}

void
destroy_cairo () {
  cairo_destroy (cr);
}
```

Eval function A function `eval` evaluates an expression and returns the value of the expression. It calls itself recursively. For example, if the node is `N_ADD`, then:

1. Calls eval(child1(node)) and gets the value1.
2. Calls eval(child2(node)) and gets the value2.
3. Returns value1+value2.

This is performed by a macro `calc` defined in the sixth line in the following program.

```
double
eval (node_t *node) {
double value = 0.0;
  if (node == NULL)
    runtime_error ("No␣expression␣to␣evaluate.\n");
#define calc(op) eval (child1(node)) op eval (child2(node))
  switch (node->type) {
    case N_EQ:
      value = (double) calc(==);
      break;
    case N_GT:
      value = (double) calc(>);
      break;
    case N_LT:
      value = (double) calc(<);
      break;
    case N_ADD:
      value = calc(+);
      break;
    case N_SUB:
      value = calc(-);
      break;
    case N_MUL:
      value = calc(*);
      break;
    case N_DIV:
      if (eval (child2(node)) == 0.0)
        runtime_error ("Division␣by␣zerp.\n");
      else
        value = calc(/);
      break;
```

```
    case N_UMINUS:
      value = -(eval (child1(node)));
      break;
    case N_ID:
      if (! (stack_lookup (name(node), &value)) && ! var_lookup (name(node), &value)
          )
        runtime_error ("Variable␣%s␣not␣defined.\n", name(node));
      break;
    case N_NUM:
      value = value(node);
      break;
    default:
      runtime_error ("Illegal␣expression.\n");
  }
  return value;
}
```

Execute function Primary procedures and procedure definitions are analyzed and executed by the function `execute`. It doesn't return any values. It calls itself recursively. The process of `N_RT` and `N_procedure_call` is complicated. It will explained after the following program. Other parts are not so difficult. Read the program below carefully so that you will understand the process.

```
/* procedure - return status */
static int proc_level = 0;
static int ret_level = 0;

void
execute (node_t *node) {
  double d, x, y;
  char *name;
  int n, i;

  if (node == NULL)
    runtime_error ("Node␣is␣NULL.\n");
  if (proc_level > ret_level)
    return;
  switch (node->type) {
    case N_program:
      execute (child1(node));
      execute (child2(node));
      break;
    case N_PU:
      pen = FALSE;
      break;
    case N_PD:
      pen = TRUE;
      break;
    case N_PW:
      line_width = eval (child1(node)); /* line width */
      break;
    case N_FD:
      d = eval (child1(node)); /* distance */
      x = d * cos (angle*M_PI/180);
      y = d * sin (angle*M_PI/180);
      /* initialize the current point = start point of the line */
      cairo_move_to (cr, cur_x, cur_y);
      cur_x += x;
      cur_y += y;
      cairo_set_line_width (cr, line_width);
      cairo_set_source_rgb (cr, fc.red, fc.green, fc.blue);
      if (pen)
        cairo_line_to (cr, cur_x, cur_y);
      else
        cairo_move_to (cr, cur_x, cur_y);
```

```
      cairo_stroke (cr);
      break;
    case N_TR:
      angle -= eval (child1(node));
      for (; angle < 0; angle += 360.0);
      for (; angle>360; angle -= 360.0);
      break;
    case N_BC:
      bc.red = eval (child1(node));
      bc.green = eval (child2(node));
      bc.blue = eval (child3(node));
#define fixcolor(c)  c = c < 0 ? 0 : (c > 1 ? 1 : c)
      fixcolor (bc.red);
      fixcolor (bc.green);
      fixcolor (bc.blue);
      /* clear the shapes and set the background color */
      cairo_set_source_rgb (cr, bc.red, bc.green, bc.blue);
      cairo_paint (cr);
      break;
    case N_FC:
      fc.red = eval (child1(node));
      fc.green = eval (child2(node));
      fc.blue = eval (child3(node));
      fixcolor (fc.red);
      fixcolor (fc.green);
      fixcolor (fc.blue);
      break;
    case N_ASSIGN:
      name = name(child1(node));
      d = eval (child2(node));
      if (! stack_replace (name, d)) /* First, tries to replace the value in the
          stack (parameter).*/
        var_replace (name, d); /* If the above fails, tries to replace the value in
            the table. If the variable isn't in the table, installs it, */
      break;
    case N_IF:
      if (eval (child1(node)))
        execute (child2(node));
      break;
    case N_RT:
      ret_level--;
      break;
    case N_RS:
      pen = TRUE;
      angle = 90.0;
      cur_x = 0.0;
      cur_y = 0.0;
      line_width = 2.0;
      fc.red = 0.0; fc.green = 0.0; fc.blue = 0.0;
      /* To change background color, use bc. */
      break;
    case N_procedure_call:
      name = name(child1(node));
node_t *proc = proc_lookup (name);
      if (! proc)
        runtime_error ("Procedure␣%s␣not␣defined.\n", name);
      if (strcmp (name, name(child1(proc))) != 0)
        runtime_error ("Unexpected␣error.␣Procedure␣%s␣is␣called,␣but␣invoked␣
            procedure␣is␣%s.\n", name, name(child1(proc)));
/* make tuples (parameter (name), argument (value)) and push them to the stack */
node_t *param_list;
node_t *arg_list;
      param_list = child2(proc);
      arg_list = child2(node);
```

```
      if (param_list == NULL) {
        if (arg_list == NULL) {
          stack_push (NULL, 0.0); /* number of argument == 0 */
        } else
          runtime_error ("Procedure␣%s␣has␣different␣number␣of␣argument␣and␣
              parameter.\n", name);
      }else {
/* Don't change the stack until finish evaluating the arguments. */
#define TEMP_STACK_SIZE 20
        char *temp_param[TEMP_STACK_SIZE];
        double temp_arg[TEMP_STACK_SIZE];
        n = 0;
        for (; param_list->type == N_parameter_list; param_list =
            child1(param_list)) {
          if (arg_list->type != N_argument_list)
            runtime_error ("Procedure␣%s␣has␣different␣number␣of␣argument␣and␣
                parameter.\n", name);
          if (n >= TEMP_STACK_SIZE)
            runtime_error ("Too␣many␣parameters.␣the␣number␣must␣be␣%d␣or␣less.\n",
                TEMP_STACK_SIZE);
          temp_param[n] = name(child2(param_list));
          temp_arg[n] = eval (child2(arg_list));
          arg_list = child1(arg_list);
          ++n;
        }
        if (param_list->type == N_ID && arg_list -> type != N_argument_list) {
          temp_param[n] = name(param_list);
          temp_arg[n] = eval (arg_list);
          if (++n >= TEMP_STACK_SIZE)
            runtime_error ("Too␣many␣parameters.␣the␣number␣must␣be␣%d␣or␣less.\n",
                TEMP_STACK_SIZE);
          temp_param[n] = NULL;
          temp_arg[n] = (double) n;
          ++n;
        } else
          runtime_error ("Unexpected␣error.\n");
        for (i = 0; i < n; ++i)
          stack_push (temp_param[i], temp_arg[i]);
      }
      ret_level = ++proc_level;
      execute (child3(proc));
      ret_level = --proc_level;
      stack_return ();
      break;
    case N_procedure_definition:
      name = name(child1(node));
      proc_install (name, node);
      break;
    case N_primary_procedure_list:
      execute (child1(node));
      execute (child2(node));
      break;
    default:
      runtime_error ("Unknown␣statement.\n");
  }
}
```

A node `N_procedure_call` is created by the parser when it has found a user defined procedure call. The procedure has been defined in the prior statement. Suppose the parser reads the following example code.

```
dp drawline (angle, distance) {
  tr angle
  fd distance
}
drawline (90, 100)
```

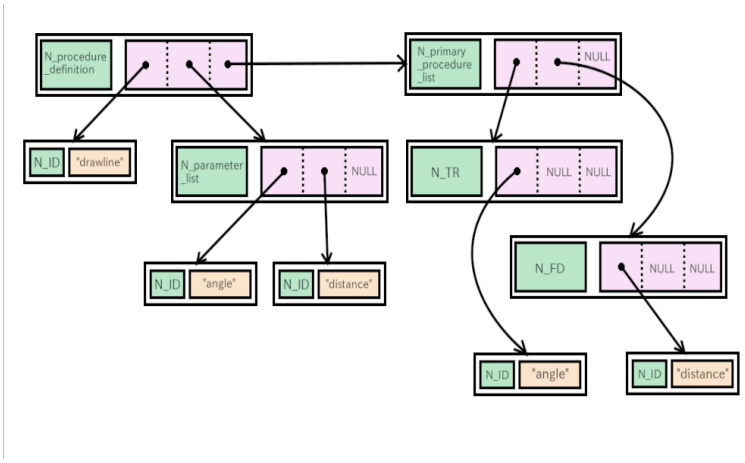


Figure 47: Nodes of drawline

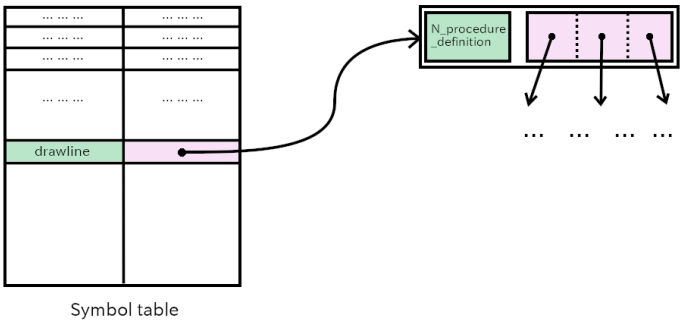


Figure 48: Symbol table

```
drawline (90, 100)
drawline (90, 100)
drawline (90, 100)
```

This example draws a square.

When The parser reads the lines from one to four, it creates nodes like this:

Runtime routine just stores the procedure to the symbol table with its name and node.

When the parser reads the fifth line in the example, it creates nodes like this:

When the runtime routine meets `N_procedure_call` node, it behaves like this:

1. Searches the symbol table for the procedure with the name.
2. Gets pointers to the node to parameters and the node to the body.
3. Creates a temporary stack. Makes a tuple of each parameter name and argument value. Pushes the tuples into the stack, and (NULL, number of parameters) finally. If no error occurs, copies them from the temporary stack to the parameter stack.
4. Increases `prc_level` by one. Sets `ret_level` to the same value as `proc_level`. `proc_level` is zero when runtime routine runs on the main routine. If it goes into a procedure, `proc_level` increases by one. Therefore, `proc_level` is the depth of the procedure call. `ret_level` is the level to return. If it is the same as `proc_level`, runtime routine executes commands in order of the commands in the procedure. If it is smaller than `proc_level`, runtime routine doesn't execute commands until it becomes the same level as `proc_level`. `ret_level` is used to return the procedure.
5. Executes the node of the body of the procedure.
6. Decreases `proc_level` by one. Sets `ret_level` to the same value as `proc_level`. Calls `stack_return`.

When the runtime routine meets `N_RT` node, it decreases `ret_level` by one so that the following commands

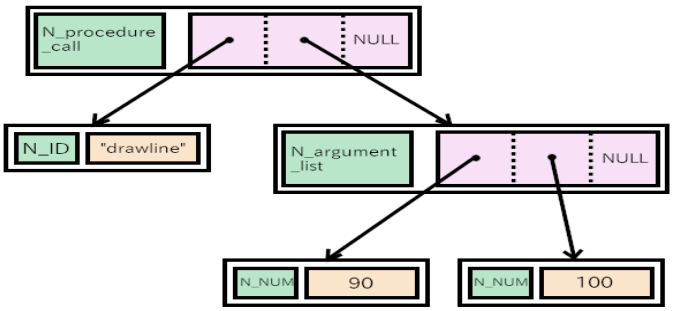


Figure 49: Nodes of procedure call

in the procedure are ignored by the runtime routine.

Runtime entry and error functions A function `run` is the entry of the runtime routine. A function `runtime_error` reports an error occurred during the runtime routine runs. (Errors which occur during the parsing are called syntax error and reported by `yyerror`.) After `runtime_error` reports an error, it stops the command execution and goes back to `run` to exit.

Setjmp and longjmp functions are used. They are declared in `<setjmp.h>`. `setjmp (buf)` saves state information in `buf` and returns zero. `longjmp(buf, 1)` restores the state information from `buf` and returns `1` (the second argument). Because the information is the status at the time `setjmp` is called, so longjmp resumes the execution at the next of setjmp function call. In the following program, longjmp resumes at the assignment to the variable `i`. When setjmp is called, 0 is assigned to `i` and `execute(node_top)` is called. On the other hand, when longjmp is called, 1 is assigned to `i` and `execute(node_top)` is not called..

`g_slist_free_full` frees all the allocated memories.

```
static jmp_buf buf;

void
run (void) {
  int i;

  if (! init_cairo()) {
    g_print ("Cairo␣not␣initialized.\n");
    return;
  }
  init_table();
  init_stack();
  ret_level = proc_level = 1;
  i = setjmp (buf);
  if (i == 0)
    execute(node_top);
  /* else ... get here by calling longjmp */
  destroy_cairo ();
  g_slist_free_full (g_steal_pointer (&list), g_free);
}

/* format supports only %s, %f and %d */
static void
runtime_error (char *format, ...) {
  va_list args;
  char *f;
  char b[3];
  char *s;
  double v;
  int i;
```

```
  va_start (args, format);
  for (f = format; *f; f++) {
    if (*f != '%') {
      b[0] = *f;
      b[1] = '\0';
      g_print ("%s", b);
      continue;
    }
    switch (*++f) {
      case 's':
        s = va_arg(args, char *);
        g_print ("%s", s);
        break;
      case 'f':
        v = va_arg(args, double);
        g_print("%f", v);
        break;
      case 'd':
        i = va_arg(args, int);
        g_print("%d", i);
        break;
      default:
        b[0] = '%';
        b[1] = *f;
        b[2] = '\0';
        g_print ("%s", b);
        break;
    }
  }
  va_end (args);

  longjmp (buf, 1);
}
```

A function `runtime_error` has a variable-length argument list.

```
void runtime_error (char *format, ...)
```

This is implemented with `<stdarg.h>` header file. The `va_list` type variable `args` will refer to each argument in turn. A function `va_start` initializes `args`. A function `va_arg` returns an argument and moves the reference of `args` to the next. A function `va_end` cleans up everything necessary at the end.

The function `runtime_error` has a similar format of printf standard function. But its format has only `%s`, `%f` and `%d`.

The functions declared in `<setjmp.h>` and `<stdarg.h>` are explained in the very famous book "The C programming language" written by Brian Kernighan and Dennis Ritchie. I referred to the book to write the program above.

The program `turtle` is unsophisticated and unpolished. If you want to make your own language, you need to know more and more. I don't know any good textbook about compilers and interpreters. If you know a good book, please let me know.

However, the following information is very useful (but old).

- Bison documentation
- Flex documentation
- Software tools written by Brian W. Kernighan & P. J. Plauger (1976)
- Unix programming environment written by Brian W. Kernighan and Rob Pike (1984)
- Source code of a language, for example, ruby.

Lately, lots of source codes are in the internet. Maybe reading source codes is the most useful for programmers.

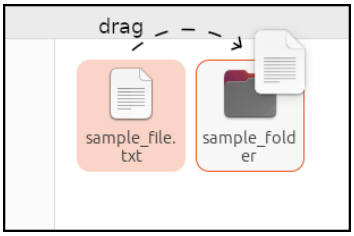


Figure 50: DND on the GUI file manager

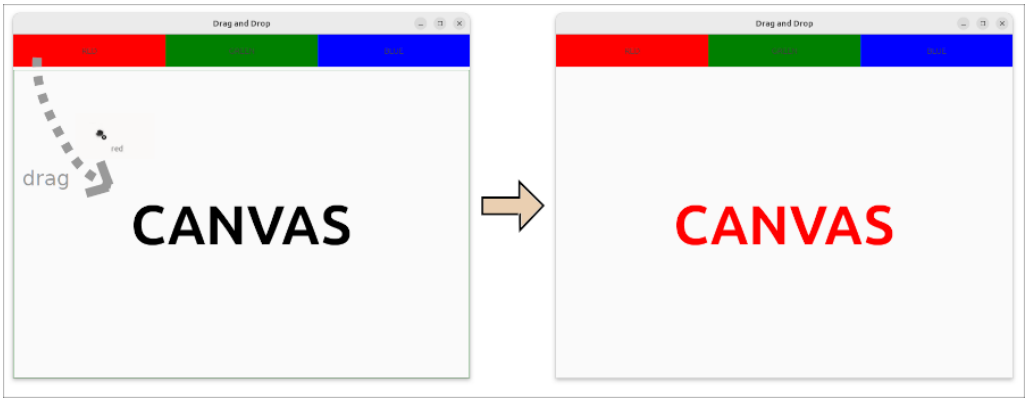


Figure 51: DND example

28 Drag and drop

28.1 What's drag and drop?

Drag and drop is also written as "Drag-and-Drop", or "DND" in short. DND is like "copy and paste" or "cut and paste". If a user drags a UI element, which is a widget, selected part or something, data is transferred from the source to the destination.

You probably have experience that you moved a file with DND.

When the DND starts, the file `sample_file.txt` is given to the system. When the DND ends, the system gives `sample_file.txt` to the directory `sample_folder` in the file manager. Therefore, it is like "cut and paste". The actual behavior may be different from the explanation here, but the concept is similar.

28.2 Example for DND

This tutorial provides a simple example in the `src/dnd` directory. It has three labels for the source and one label for the destination. The source labels have "red", "green" or "blue" labels. If a user drags the label to the destination label, the font color will be changed.

28.3 UI file

The widgets are defined in the XML file `dnd.ui`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <object class="GtkApplicationWindow" id="win">
4      <property name="default-width">800</property>
5      <property name="default-height">600</property>
6      <property name="resizable">FALSE</property>
7      <property name="title">Drag and Drop</property>
8      <child>
9        <object class="GtkBox">
10         <property name="hexpand">TRUE</property>
11         <property name="vexpand">TRUE</property>
12         <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
```

```
        <property name="spacing">5</property>
        <child>
          <object class="GtkBox">
            <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
            <property name="homogeneous">TRUE</property>
            <child>
              <object class="GtkLabel" id="red">
                <property name="label">RED</property>
                <property name="justify">GTK_JUSTIFY_CENTER</property>
                <property name="name">red</property>
              </object>
            </child>
            <child>
              <object class="GtkLabel" id="green">
                <property name="label">GREEN</property>
                <property name="justify">GTK_JUSTIFY_CENTER</property>
                <property name="name">green</property>
              </object>
            </child>
            <child>
              <object class="GtkLabel" id="blue">
                <property name="label">BLUE</property>
                <property name="justify">GTK_JUSTIFY_CENTER</property>
                <property name="name">blue</property>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkLabel" id="canvas">
            <property name="label">CANVAS</property>
            <property name="justify">GTK_JUSTIFY_CENTER</property>
            <property name="name">canvas</property>
            <property name="hexpand">TRUE</property>
            <property name="vexpand">TRUE</property>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

It is converted to a resource file by `glib-compile-resources`. The compiler uses an XML file `dnd.gresource.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/com/github/ToshioCP/dnd">
    <file>dnd.ui</file>
  </gresource>
</gresources>
```

28.4 C file dnd.c

The C file `dnd.c` isn't a big file. The number of the lines is less than a hundred. A GtkApplication object is created in the function `main`.

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

  app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
  g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
```

```
8     g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
9     stat =g_application_run (G_APPLICATION (app), argc, argv);
10    g_object_unref (app);
11    return stat;
12  }
```

The application ID is defined as:

```
#define APPLICATION_ID "com.github.ToshioCP.dnd"
```

28.4.1 Startup signal handler

Most of the work is done in the "startup" signal handler.

Two objects GtkDragSource and GtkDropTarget is used for DND implementation.

- Drag source: A drag source (GtkDragSource instance) is an event controller. It initiates a DND operation when the user clicks and drags the widget. If a data, in the form of GdkContentProvider, is set in advance, it gives the data to the system at the beginning of the drag.
- Drop target: A drop target (GtkDropTarget) is also an event controller. You can get the data in the GtkDropTarget::drop signal handler.

The example below uses these objects in a very simple way. You can use number of features that the two objects have. See the following links for more information.

- Drag-and-Drop in GTK
- GtkDragSource
- GtkDropTarget

```
1   static void
2   app_startup (GApplication *application) {
3     GtkApplication *app = GTK_APPLICATION (application);
4     GtkBuilder *build;
5     GtkWindow *win;
6     GtkLabel *src_labels[3];
7     int i;
8     GtkLabel *canvas;
9     GtkDragSource *src;
10    GdkContentProvider* content;
11    GtkDropTarget *tgt;
12    GdkDisplay *display;
13    char *s;
14
15    build = gtk_builder_new_from_resource ("/com/github/ToshioCP/dnd/dnd.ui");
16    win = GTK_WINDOW (gtk_builder_get_object (build, "win"));
17    src_labels[0] = GTK_LABEL (gtk_builder_get_object (build, "red"));
18    src_labels[1] = GTK_LABEL (gtk_builder_get_object (build, "green"));
19    src_labels[2] = GTK_LABEL (gtk_builder_get_object (build, "blue"));
20    canvas = GTK_LABEL (gtk_builder_get_object (build, "canvas"));
21    gtk_window_set_application (win, app);
22    g_object_unref (build);
23
24    for (i=0; i<3; ++i) {
25      src = gtk_drag_source_new ();
26      content = gdk_content_provider_new_typed (G_TYPE_STRING, gtk_widget_get_name
          (GTK_WIDGET (src_labels[i])));
27      gtk_drag_source_set_content (src, content);
28      g_object_unref (content);
29      gtk_widget_add_controller (GTK_WIDGET (src_labels[i]), GTK_EVENT_CONTROLLER
          (src)); // The ownership of src is taken by the instance.
30    }
31
32    tgt = gtk_drop_target_new (G_TYPE_STRING, GDK_ACTION_COPY);
33    g_signal_connect (tgt, "drop", G_CALLBACK (drop_cb), NULL);
34    gtk_widget_add_controller (GTK_WIDGET (canvas), GTK_EVENT_CONTROLLER (tgt)); //
        The ownership of tgt is taken by the instance.
```

```
35
36    provider = gtk_css_provider_new ();
37    s = g_strdup_printf (format, "black");
38    gtk_css_provider_load_from_data (provider, s, -1);
39    g_free (s);
40    display = gdk_display_get_default ();
41    gtk_style_context_add_provider_for_display (display, GTK_STYLE_PROVIDER (provider),
42                                                GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
43    g_object_unref (provider); // The provider is still alive because the display owns
          it.
44  }
```

- 15-22: Builds the widgets. The array `source_labels[]` points the source labels red, green and blue in the ui file. The variable `canvas` points the destination label.
- 24-30: Sets the DND source widgets. The for-loop carries out through the array `src_labels[]` each of which points the source widget, red, green or blue label.
- 25: Creates a new GtkDragSource instance.
- 26: Creates a new GdkContentProvider instance with the string “red”, “green” or “blue. They are the name of the widgets. These strings are the data to transfer through the DND operation.
- 27: Sets the content of the drag source to the GdkContentProvider instance above.
- 28: Content is useless so it is destroyed.
- 29: Add the event controller, which is actually the drag source, to the widget. If a DND operation starts on the widget, the corresponding drag source works and the data is given to the system.
- 32-34: Sets the DND drop target.
- 32: Creates a new GtkDropTarget instance. The first parameter is the GType of the data. The second parameter is a GdkDragAction enumerate constant. The arguments here are string type and the constant for copy.
- 33: Connects the “drop” signal and the handler `drop_cb`.
- 34: Add the event controller, which is actually the drop target, to the widget.
- 36-43: Sets CSS.
- 37: A varable `format` is static and defined at the top of the program. Static variables are shown below.

```
static GtkCssProvider *provider = NULL;
static const char *format = "label␣{padding:␣20px;}␣label#red␣{background:␣red;}␣"
  "label#green␣{background:␣green;}␣label#blue␣{background:␣blue;}␣"
  "label#canvas␣{color:␣%s;␣font-weight:␣bold;␣font-size:␣72pt;}";
```

28.4.2 Activate signal handler

```
1  static void
2  app_activate (GApplication *application) {
3    GtkApplication *app = GTK_APPLICATION (application);
4    GtkWindow *win;
5
6    win = gtk_application_get_active_window (app);
7    gtk_window_present (win);
8  }
```

This handler just shows the window.

28.4.3 Drop signal handler

```
1  static gboolean
2  drop_cb (GtkDropTarget* self, const GValue* value, gdouble x, gdouble y, gpointer
       user_data) {
3    char *s;
4
5    s = g_strdup_printf (format, g_value_get_string (value));
6    gtk_css_provider_load_from_data (provider, s, -1);
7    g_free (s);
8    return TRUE;
9  }
```

The “drop” signal handler has five parameters.

- GtkDropTarget instance on which the signal has been emitted.
- GValue that holds the data from the source.
- The arguments `x` and `y` are the coordinate of the mouse when released.
- User data was set when the signal and handler was connected.

The string from the GValue is “red”, “green” or “blue”. It replaces “%s” in the variable `format`. That means the font color of the label `canvas` will turn to the color.

28.5 Meson.build

The file `meson.build` controls the building process.

```
1 project('dnd', 'c')
2
3 gtkdep = dependency('gtk4')
4
5 gnome = import('gnome')
6 resources = gnome.compile_resources('resources','dnd.gresource.xml')
7
8 executable(meson.project_name(), 'dnd.c', resources, dependencies: gtkdep,
      export_dynamic: true, install: false)
```

You can build it from the command line.

```
$ cd src/dnd
$ meson setup _build
$ ninja -C _build
$ _build/dnd
```

The source files are under the directory `src/dnd` of the repository. Download it and see the directory.

29 GtkListView

GTK 4 has added new list objects GtkListView, GtkGridView and GtkColumnView. The new feature is described in Gtk API Reference – List Widget Overview.

GTK 4 has other means to implement lists. They are GtkListBox and GtkTreeView which are took over from GTK 3. There’s an article in Gtk Development blog about list widgets by Matthias Clasen. He described why GtkListView are developed to replace GtkTreeView. GtkTreeView is deprecated since version 4.10.

GtkListView, GtkGridView, GtkColumnView and related objects are described in Section 29 to 33.

29.1 Outline

A list is a sequential data structure. For example, an ordered string sequence “one”, “two”, “three”, “four” is a list. Each element is called item. A list is like an array, but in many cases it is implemented with pointers which point to the next items of the list. And it has a start point. So, each item can be referred by the index of the item (first item, second item, …, nth item, …). There are two cases. The one is the index starts from one (one-based) and the other is from zero (zero-based).

Gio provides GListModel interface. It is a zero-based list and its items are the same type of GObject descendants, or objects that implement the same interface. An object implements GListModel is not a widget. So, the list is not displayed on the screen directly. There’s another object GtkListView which is a widget to display the list. The items in the list need to be connected to the items in GtkListView. GtkListItemFactory instance maps items in the list to GtkListView.

29.2 GListModel and GtkStringList

If you want to make a list of strings with GListModel, for example, “one”, “two”, “three”, “four”, note that strings can’t be items of the list. Because GListModel is a list of GObject objects and strings aren’t GObject

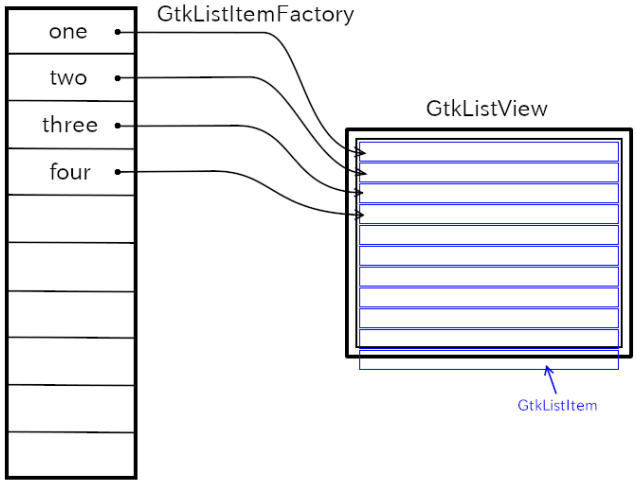


Figure 52: List

objects. The word “GObject” here means “GObject class or its descendant class”. So, you need a wrapper which is a GObject and contains a string. GtkStringObject is the wrapper object and GtkStringList, implements GListModel, is a list of GtkStringObject.

```
char *array[] = {"one", "two", "three", "four", NULL};
GtkStringList *stringlist = gtk_string_list_new ((const char * const *) array);
```

The function `gtk_string_list_new` creates a GtkStringList object. Its items are GtkStringObject objects which contain the strings “one”, “two”, “three” and “four”. There are functions to add items to the list or remove items from the list.

- `gtk_string_list_append` appends an item to the list
- `gtk_string_list_remove` removes an item from the list
- `gtk_string_list_get_string` gets a string in the list

See GTK 4 API Reference – GtkStringList for further information.

Other list objects will be explained later.

29.3 GtkSelectionModel

GtkSelectionModel is an interface to support for selections. Thanks to this model, user can select items by clicking on them. It is implemented by GtkMultiSelection, GtkNoSelection and GtkSingleSelection objects. These three objects are usually enough to build an application. They are created with another GListModel. You can also create them alone and add a GListModel later.

- GtkMultiSelection supports multiple selection.
- GtkNoSelection supports no selection. This is a wrapper to GListModel when GtkSelectionModel is needed.
- GtkSingleSelection supports single selection.

29.4 GtkListView

GtkListView is a widget to show GListModel items. GtkListItem is used by GtkListView to represent items of a list model. But, GtkListItem itself is not a widget, so a user needs to set a widget, for example GtkLabel, as a child of GtkListItem to display an item of the list model. “item” property of GtkListItem points an object that belongs to the list model.

In case the number of items is very big, for example more than a thousand, GtkListItem is recycled and connected to another item which is newly displayed. This recycle makes the number of GtkListItem objects

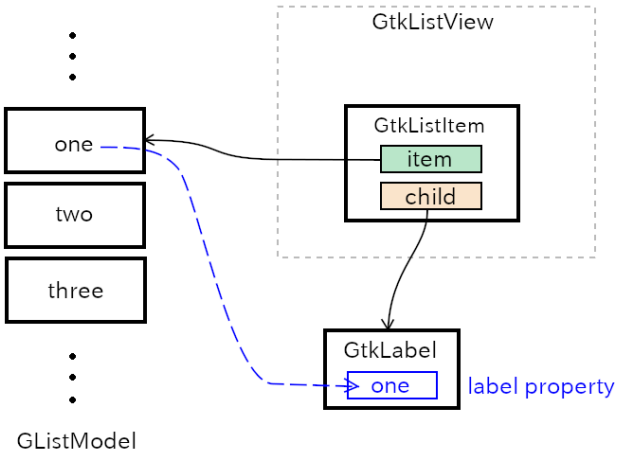


Figure 53: GtkListItem

fairly small, less than 200. This is very effective to restrain the growth of memory consumption so that GListModel can contain lots of items, for example, more than a million items.

29.5 GtkListItemFactory

GtkListItemFactory creates or recycles GtkListItem and connects it with an item of the list model. There are two child classes of this factory, GtkSignalListItemFactory and GtkBuilderListItemFactory.

29.5.1 GtkSignalListItemFactory

GtkSignalListItemFactory provides signals for users to configure a GtkListItem object. There are four signals.

1. "setup" is emitted to set up GtkListItem object. A user sets its child widget in the handler. For example, creates a GtkLabel widget and sets the child property of the GtkListItem to it. This setting is kept even the GtkListItem instance is recycled (to bind to another item of GListModel).
2. "bind" is emitted to bind an item in the list model to the widget. For example, a user gets the item from "item" property of the GtkListItem instance. Then gets the string of the item and sets the label property of the GtkLabel instance with the string. This signal is emitted when the GtkListItem is newly created, recycled or some changes has happened to the item of the list.
3. "unbind" is emitted to unbind an item. A user undoes everything done in step 2 in the signal handler. If some object are created in step 2, they must be destroyed.
4. "teardown" is emitted to undo everything done in step 1. So, the widget created in step 1 must be destroyed. After this signal, the list item will be destroyed.

The following program `list1.c` shows the list of strings "one", "two", "three" and "four". GtkNoSelection is used, so user can't select any item.

```
1  #include <gtk/gtk.h>
2
3  static void
4  setup_cb (GtkSignalListItemFactory *self, GtkListItem *listitem, gpointer user_data)
       {
5    GtkWidget *lb = gtk_label_new (NULL);
6    gtk_list_item_set_child (listitem, lb);
7    /* Because gtk_list_item_set_child sunk the floating reference of lb, releasing
         (unref) isn't necessary for lb. */
8  }
9
10 static void
```

```
11 bind_cb (GtkSignalListItemFactory *self, GtkListItem *listitem, gpointer user_data) {
12   GtkWidget *lb = gtk_list_item_get_child (listitem);
13   /* Strobj is owned by the instance. Caller mustn't change or destroy it. */
14   GtkStringObject *strobj = gtk_list_item_get_item (listitem);
15   /* The string returned by gtk_string_object_get_string is owned by the instance. */
16   gtk_label_set_text (GTK_LABEL (lb), gtk_string_object_get_string (strobj));
17 }
18
19 static void
20 unbind_cb (GtkSignalListItemFactory *self, GtkListItem *listitem, gpointer
       user_data) {
21   /* There's nothing to do here. */
22 }
23
24 static void
25 teardown_cb (GtkSignalListItemFactory *self, GtkListItem *listitem, gpointer
       user_data) {
26   /* There's nothing to do here. */
27   /* GtkListItem instance will be destroyed soon. You don't need to set the child to
         NULL. */
28 }
29
30 static void
31 app_activate (GApplication *application) {
32   GtkApplication *app = GTK_APPLICATION (application);
33   GtkWidget *win = gtk_application_window_new (app);
34   gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
35   GtkWidget *scr = gtk_scrolled_window_new ();
36   gtk_window_set_child (GTK_WINDOW (win), scr);
37
38   char *array[] = {
39     "one", "two", "three", "four", NULL
40   };
41   /* sl is owned by ns */
42   /* ns and factory are owned by lv. */
43   /* Therefore, you don't need to care about their destruction. */
44   GtkStringList *sl =  gtk_string_list_new ((const char * const *) array);
45   GtkNoSelection *ns =  gtk_no_selection_new (G_LIST_MODEL (sl));
46
47   GtkListItemFactory *factory = gtk_signal_list_item_factory_new ();
48   g_signal_connect (factory, "setup", G_CALLBACK (setup_cb), NULL);
49   g_signal_connect (factory, "bind", G_CALLBACK (bind_cb), NULL);
50   /* The following two lines can be left out. The handlers do nothing. */
51   g_signal_connect (factory, "unbind", G_CALLBACK (unbind_cb), NULL);
52   g_signal_connect (factory, "teardown", G_CALLBACK (teardown_cb), NULL);
53
54   GtkWidget *lv = gtk_list_view_new (GTK_SELECTION_MODEL (ns), factory);
55   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), lv);
56   gtk_window_present (GTK_WINDOW (win));
57 }
58
59 /* ----- main ----- */
60 #define APPLICATION_ID "com.github.ToshioCP.list1"
61
62 int
63 main (int argc, char **argv) {
64   GtkApplication *app;
65   int stat;
66
67   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
68
69   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
70
71   stat =g_application_run (G_APPLICATION (app), argc, argv);
```

Figure 54: list1

```
72    g_object_unref (app);
73    return stat;
74  }
```

The file `list1.c` is located under the directory src/misc. Make a shell script below and save it to your bin directory, for example `$HOME/bin`.

```
gcc `pkg-config --cflags gtk4` $1.c `pkg-config --libs gtk4`
```

Change the current directory to the directory includes `list1.c` and type as follows.

```
$ chmod 755 $HOME/bin/comp # or chmod 755 (your bin directory)/comp
$ comp list1
$ ./a.out
```

Then, the following window appears.

The program is not so difficult. If you feel some difficulty, read this section again, especially GtkSignalListItemFactory subsubsection.

29.5.2 GtkBuilderListItemFactory

GtkBuilderListItemFactory is another GtkListItemFactory. Its behavior is defined with ui file.

```
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkLabel">
        <binding name="label">
          <lookup name="string" type="GtkStringObject">
            <lookup name="item">GtkListItem</lookup>
          </lookup>
        </binding>
      </object>
    </property>
  </template>
</interface>
```

Template tag is used to define GtkListItem. And its child property is GtkLabel object. The factory sees this template and creates GtkLabel and sets the child property of GtkListItem. This is the same as what setup handler of GtkSignalListItemFactory did.

Then, bind the label property of the GtkLabel to the string property of a GtkStringObject. The string object refers to the item property of the GtkListItem. So, the lookup tag is like this:

```
label <- string <- GtkStringObject <- item <- GtkListItem
```

The last lookup tag has a content `GtkListItem`. Usually, C type like `GtkListItem` doesn't appear in the content of tags. This is a special case. There is an explanation in the GTK Development Blog by Matthias Clasen.

Remember that the classname (GtkListItem) in a ui template is used as the “this” pointer referring to the object that is being instantiated.

Therefore, GtkListItem instance is used as the `this` object of the lookup tag when it is evaluated. `this` object will be explained in section 31.

The C source code is as follows. Its name is `list2.c` and located under src/misc directory.

```
1  #include <gtk/gtk.h>
2
3  static void
4  app_activate (GApplication *application) {
5    GtkApplication *app = GTK_APPLICATION (application);
6    gtk_window_present (gtk_application_get_active_window(app));
7  }
8
9  static void
10 app_startup (GApplication *application) {
11   GtkApplication *app = GTK_APPLICATION (application);
12   GtkWidget *win = gtk_application_window_new (app);
13   gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
14   GtkWidget *scr = gtk_scrolled_window_new ();
15   gtk_window_set_child (GTK_WINDOW (win), scr);
16
17   char *array[] = {
18     "one", "two", "three", "four", NULL
19   };
20   GtkStringList *sl = gtk_string_list_new ((const char * const *) array);
21   GtkSingleSelection *ss = gtk_single_selection_new (G_LIST_MODEL (sl));
22
23   const char *ui_string =
24 "<interface>"
25   "<template␣class=\"GtkListItem\">"
26     "<property␣name=\"child\">"
27       "<object␣class=\"GtkLabel\">"
28         "<binding␣name=\"label\">"
29           "<lookup␣name=\"string\"␣type=\"GtkStringObject\">"
30             "<lookup␣name=\"item\">GtkListItem</lookup>"
31           "</lookup>"
32         "</binding>"
33       "</object>"
34     "</property>"
35   "</template>"
36 "</interface>"
37 ;
38   GBytes *gbytes = g_bytes_new_static (ui_string, strlen (ui_string));
39   GtkListItemFactory *factory = gtk_builder_list_item_factory_new_from_bytes (NULL,
       gbytes);
40
41   GtkWidget *lv = gtk_list_view_new (GTK_SELECTION_MODEL (ss), factory);
42   gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), lv);
43 }
44
45 /* ----- main ----- */
46 #define APPLICATION_ID "com.github.ToshioCP.list2"
47
48 int
49 main (int argc, char **argv) {
50   GtkApplication *app;
51   int stat;
52
53   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
54
55   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
56   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
```

```
57
58    stat =g_application_run (G_APPLICATION (app), argc, argv);
59    g_object_unref (app);
60    return stat;
61  }
```

No signal handler is needed for GtkBulderListItemFactory. GtkSingleSelection is used, so user can select one item at a time.

Because this is a small program, the ui data is given as a string.

29.6 GtkDirectoryList

GtkDirectoryList is a list model containing GFileInfo objects which are information of files under a certain directory. It uses `g_file_enumerate_children_async()` to get the GFileInfo objects. The list model is created by `gtk_directory_list_new` function.

```
GtkDirectoryList *gtk_directory_list_new (const char *attributes, GFile *file);
```

`attributes` is a comma separated list of file attributes. File attributes are key-value pairs. A key consists of a namespace and a name. For example, "standard::name" key is the name of a file. "standard" means general file information. "name" means filename. The following table shows some example.

key	meaning
standard::type	file type. for example, regular file, directory, symbolic link, etc.
standard::name	filename
standard::size	file size in bytes
access::can-read	read privilege if the user is able to read the file
time::modified	the time the file was last modified in seconds since the UNIX epoch

The current directory is ".". The following program makes GtkDirectoryList `dl` and its contents are GFileInfo objects under the current directory.

```
GFile *file = g_file_new_for_path (".");
GtkDirectoryList *dl = gtk_directory_list_new ("standard::name", file);
g_object_unref (file);
```

It is not so difficult to make file listing program by changing `list2.c` in the previous subsection. One problem is that GInfoFile doesn't have properties. Lookup tag look for a property, so it is useless for looking for a filename from a GFileInfo object. Instead, closure tag is appropriate in this case. Closure tag specifies a function and the type of the return value of the function.

```
char *
get_file_name (GtkListItem *item, GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_strdup (g_file_info_get_name (info)) : NULL;
}
... ...
... ...

"<interface>"
  "<template␣class=\"GtkListItem\">"
    "<property␣name=\"child\">"
      "<object␣class=\"GtkLabel\">"
        "<binding␣name=\"label\">"
          "<closure␣type=\"gchararray\"␣function=\"get_file_name\">"
            "<lookup␣name=\"item\">GtkListItem</lookup>"
          "</closure>"
        "</binding>"
      "</object>"
    "</property>"
  "</template>"
"</interface>"
```

- The string “gchararray” is a type name. The type “gchar” is a type name and it is the same as C type “char”. Therefore, “gchararray” is “an array of char type”, which is the same as string type. It is used to get the type of GValue object. GValue is a generic value and it can contain various type of values. For example, the type name can be gboolean, gchar (char), gint (int), gfloat (float), gdouble (double), gchararray (char *) and so on. These type names are the names of the fundamental types that are registered to the type system. See GObject tutorial.
- Closure tag has type attribute and function attribute. Function attribute specifies the function name and type attribute specifies the type of the return value of the function. The contents of closure tag (it is between <closure…> and</closure>) is parameters of the function. `<lookup name="item">GtkListItem</lookup>` gives the value of the item property of the GtkListItem. This will be the second argument of the function. The first parameter is always the GListItem instance, which is a ‘this’ object. The ‘this’ object is explained in section 31.
- `gtk_file_name` function is the callback function for the closure tag. It first checks the `info` parameter. Because it can be NULL when GListItem `item` is unbounded. If it’s GFileInfo, it returns the copied filename. Because the return value (filename) of `g_file_info_get_name` is owned by GFileInfo object. So, the the string needs to be duplicated to give the ownership to the caller. Binding tag binds the “label” property of the GtkLabel to the closure tag.

The whole program (`list3.c`) is as follows. The program is located in src/misc directory.

```
1  #include <gtk/gtk.h>
2
3  char *
4  get_file_name (GtkListItem *item, GFileInfo *info) {
5    return G_IS_FILE_INFO (info) ? g_strdup (g_file_info_get_name (info)) : NULL;
6  }
7
8  static void
9  app_activate (GApplication *application) {
10   GtkApplication *app = GTK_APPLICATION (application);
11   gtk_window_present (gtk_application_get_active_window(app));
12 }
13
14 static void
15 app_startup (GApplication *application) {
16   GtkApplication *app = GTK_APPLICATION (application);
17   GtkWidget *win = gtk_application_window_new (app);
18   gtk_window_set_default_size (GTK_WINDOW (win), 600, 400);
19   GtkWidget *scr = gtk_scrolled_window_new ();
20   gtk_window_set_child (GTK_WINDOW (win), scr);
21
22   GFile *file = g_file_new_for_path (".");
23   GtkDirectoryList *dl = gtk_directory_list_new ("standard::name", file);
24   g_object_unref (file);
25   GtkNoSelection *ns = gtk_no_selection_new (G_LIST_MODEL (dl));
26
27   const char *ui_string =
28 "<interface>"
29   "<template␣class=\"GtkListItem\">"
30     "<property␣name=\"child\">"
31       "<object␣class=\"GtkLabel\">"
32         "<binding␣name=\"label\">"
33           "<closure␣type=\"gchararray\"␣function=\"get_file_name\">"
34             "<lookup␣name=\"item\">GtkListItem</lookup>"
35           "</closure>"
36         "</binding>"
37       "</object>"
38     "</property>"
39   "</template>"
40 "</interface>"
41 ;
42   GBytes *gbytes = g_bytes_new_static (ui_string, strlen (ui_string));
43   GtkListItemFactory *factory = gtk_builder_list_item_factory_new_from_bytes (NULL,
       gbytes);
```

```
44
45    GtkWidget *lv = gtk_list_view_new (GTK_SELECTION_MODEL (ns), factory);
46    gtk_scrolled_window_set_child (GTK_SCROLLED_WINDOW (scr), lv);
47  }
48
49  /* ----- main ----- */
50  #define APPLICATION_ID "com.github.ToshioCP.list3"
51
52  int
53  main (int argc, char **argv) {
54    GtkApplication *app;
55    int stat;
56
57    app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
58
59    g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
60    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
61
62    stat =g_application_run (G_APPLICATION (app), argc, argv);
63    g_object_unref (app);
64    return stat;
65  }
```

The ui data (xml data above) is used to build the GListItem template at runtime. GtkBuilder refers to the symbol table to find the function `get_file_name`.

Generally, a symbol table is used by a linker to link objects to an executable file. It includes function names and their location. A linker usually doesn't put a symbol table into the created executable file. But if `--export-dynamic` option is given, the linker adds the symbol table to the executable file.

To accomplish it, an option `-Wl,--export-dynamic` is given to the C compiler.

- `-Wl` is a C compiler option that passes the following option to the linker.
- `--export-dynamic` is a linker option. The following is cited from the linker document. "When creating a dynamically linked executable, add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols which are visible from dynamic objects at run time."

Compile and execute it.

```
$ gcc -Wl,--export-dynamic `pkg-config --cflags gtk4` list3.c `pkg-config --libs
    gtk4`
```

You can also make a shell script to compile `list3.c`

```
gcc -Wl,--export-dynamic `pkg-config --cflags gtk4` $1.c `pkg-config --libs gtk4`
```

Save this one liner to a file `comp`. Then, copy it to `$HOME/bin` and give it executable permission.

```
$ cp comp $HOME/bin/comp
$ chmod +x $HOME/bin/comp
```

You can compile `list3.c` and execute it, like this:

```
$ comp list3
$ ./a.out
```

30 GtkGridView and activate signal

GtkGridView is similar to GtkListView. It displays a GListModel as a grid, which is like a square tessellation.

This is often seen when you use a file browser like GNOME Files (Nautilus).

In this section, let's make a very simple file browser `list4`. It just shows the files in the current directory. And a user can choose list or grid by clicking on buttons in the tool bar. Each item in the list or grid has an icon and a filename. In addition, `list4` provides the way to open the `tfe` text editor to show a text file. A user can do that by double clicking on an item or pressing enter key when an item is selected.

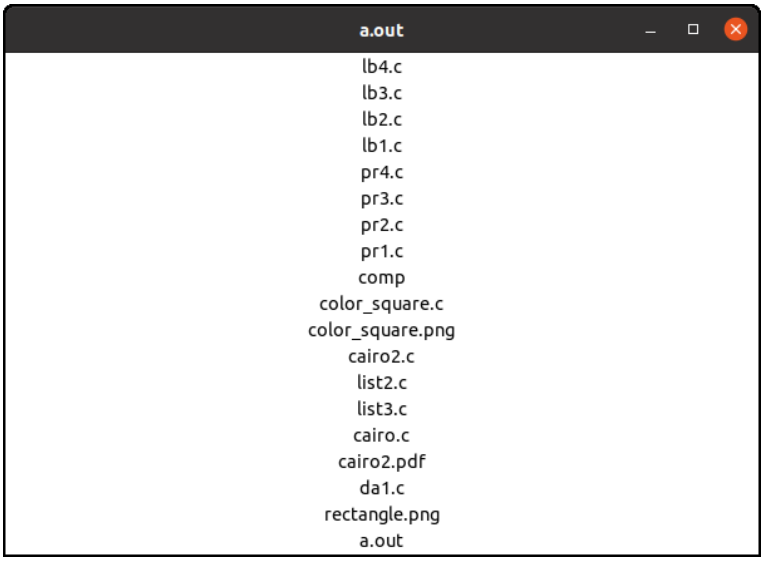


Figure 55: screenshot list3

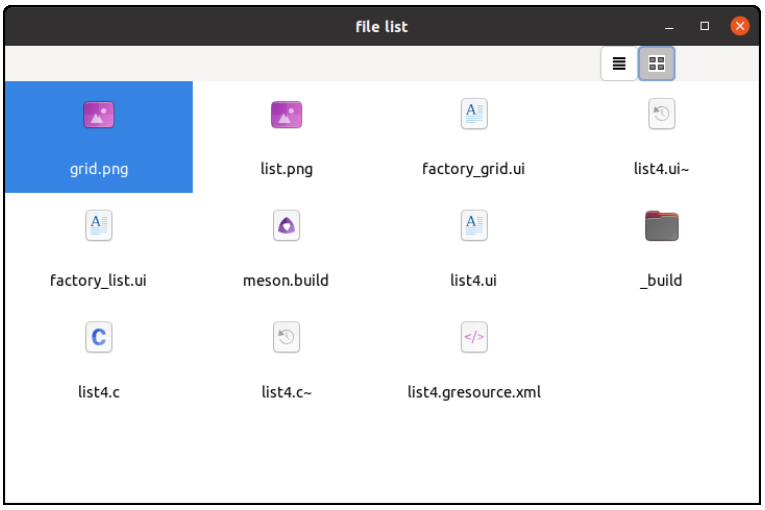


Figure 56: Grid

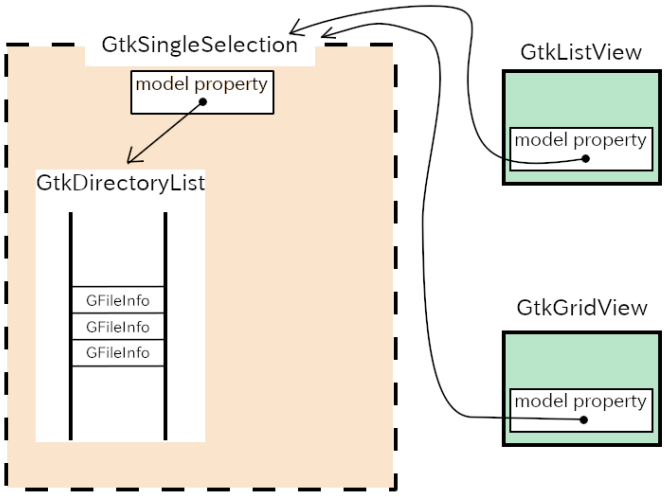


Figure 57: DirectoryList

30.1 GtkDirectoryList

GtkDirectoryList implements GListModel and it contains information of files in a certain directory. The items of the list are GFileInfo objects.

In the `list4` source files, GtkDirectoryList is described in a ui file and built by GtkBuilder. The GtkDirectoryList instance is assigned to the "model" property of a GtkSingleSelection instance. And the GtkSingleSelection instance is assigned to the "model" property of a GListView or GGridView instance.

```
GtkListView (model property) => GtkSingleSelection (model property) =>
    GtkDirectoryList
GtkGridView (model property) => GtkSingleSelection (model property) =>
    GtkDirectoryList
```

The following is a part of the ui file `list4.ui`.

```
<object class="GtkListView" id="list">
  <property name="model">
    <object class="GtkSingleSelection" id="singleselection">
      <property name="model">
        <object class="GtkDirectoryList" id="directorylist">
          <property
              name="attributes">standard::name,standard::icon,standard::content-type</property>
        </object>
      </property>
    </object>
  </property>
</object>
<object class="GtkGridView" id="grid">
  <property name="model">singleselection</property>
</object>
```

GtkDirectoryList has an "attributes" property. It is attributes of GFileInfo such as "standard::name", "standard::icon" and "standard::content-type".

- standard::name is a filename.
- standard::icon is an icon of the file. It is a GIcon object.
- standard::content-type is a content-type. Content-type is the same as mime type for the internet. For example, "text/plain" is a text file, "text/x-csrc" is a C source code and so on. ("text/x-csrc"is not registered to IANA media types. Such "x-" subtype is not a standard mime type.) Content type is also used by desktop systems.

GtkGridView uses the same GtkSingleSelection instance (`singleselection`). So, its model property is set to it.

30.2 Ui file of the window

The window is built with the following ui file. (See the screenshot at the beginning of this section).

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <object class="GtkApplicationWindow" id="win">
    <property name="title">file list</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
    <child>
      <object class="GtkBox" id="boxv">
        <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
        <child>
          <object class="GtkBox" id="boxh">
            <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
            <child>
              <object class="GtkLabel" id="dmy1">
                <property name="hexpand">TRUE</property>
              </object>
            </child>
            <child>
              <object class="GtkButton" id="btnlist">
                <property name="name">btnlist</property>
                <property name="action-name">win.view</property>
                <property name="action-target">&apos;list&apos;</property>
                <child>
                  <object class="GtkImage">
                    <property
                        name="resource">/com/github/ToshioCP/list4/list.png</property>
                  </object>
                </child>
              </object>
            </child>
            <child>
              <object class="GtkButton" id="btngrid">
                <property name="name">btngrid</property>
                <property name="action-name">win.view</property>
                <property name="action-target">&apos;grid&apos;</property>
                <child>
                  <object class="GtkImage">
                    <property
                        name="resource">/com/github/ToshioCP/list4/grid.png</property>
                  </object>
                </child>
              </object>
            </child>
            <child>
              <object class="GtkLabel" id="dmy2">
                <property name="width-chars">10</property>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkScrolledWindow" id="scr">
            <property name="hexpand">TRUE</property>
            <property name="vexpand">TRUE</property>
          </object>
        </child>
      </object>
```

```
56      </child>
57    </object>
58    <object class="GtkListView" id="list">
59      <property name="model">
60        <object class="GtkSingleSelection" id="singleselection">
61          <property name="model">
62            <object class="GtkDirectoryList" id="directory_list">
63              <property
                    name="attributes">standard::name,standard::icon,standard::content-type</property>
64            </object>
65          </property>
66        </object>
67      </property>
68    </object>
69    <object class="GtkGridView" id="grid">
70      <property name="model">singleselection</property>
71    </object>
72  </interface>
```

The file consists of two parts. The first part begins at the line 3 and ends at line 57. This part is the widgets from the top level window to the scrolled window. It also includes two buttons. The second part begins at line 58 and ends at line 71. This is the part of GtkListView and GtkGridView.

- 13-17, 42-46: Two labels are dummy labels. They just work as a space to put the two buttons at the appropriate position.
- 18-41: GtkButton `btnlist` and `btngrid`. These two buttons work as selection buttons to switch from list to grid and vice versa. These two buttons are connected to a stateful action `win.view`. This action has a parameter. Such action consists of prefix, action name and parameter. The prefix of the action is `win`, which means the action belongs to the top level window. The prefix gives the scope of the action. The action name is `view`. The parameters are `list` or `grid`, which show the state of the action. A parameter is also called a target, because it is a target to which the action changes its state. We often write the detailed action like "win.view::list" or "win.view::grid".
- 21-22: The properties "action-name" and "action-target" belong to GtkActionable interface. GtkButton implements GtkActionable. The action name is "win.view" and the target is "list". Generally, a target is GVariant, which can be string, integer, float and so on. You need to use GVariant text format to write GVariant value in ui files. If the type of the GVariant value is string, then the value with GVariant text format is bounded by single quotes or double quotes. Because ui file is xml format text, single quote cannot be written without escape. Its escape sequence is '. Therefore, the target 'list' is written as 'list'. Because the button is connected to the action, "clicked" signal handler isn't needed.
- 23-27: The child widget of the button is GtkImage. GtkImage has a "resource" property. It is a GResource and GtkImage reads an image data from the resource and sets the image. This resource is built from 24x24-sized png image data, which is an original icon.
- 50-53: GtkScrolledWindow. Its child widget will be set with GtkListView or GtkGridView.

The action `view` is created, connected to the "activate" signal handler and inserted to the window (action map) as follows.

```
act_view = g_simple_action_new_stateful ("view", g_variant_type_new("s"),
    g_variant_new_string ("list"));
g_signal_connect (act_view, "activate", G_CALLBACK (view_activated), NULL);
g_action_map_add_action (G_ACTION_MAP (win), G_ACTION (act_view));
```

The signal handler `view_activated` will be explained later.

30.3 Factories

Each view (GtkListView and GtkGridView) has its own factory because its items have different structure of widgets. The factories are GtkBuilderListItemFactory objects. Their ui files are as follows.

factory_list.ui

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
```

```
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkBox">
        <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
        <property name="spacing">20</property>
        <child>
          <object class="GtkImage">
            <binding name="gicon">
              <closure type="GIcon" function="get_icon">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
        <child>
          <object class="GtkLabel">
            <property name="hexpand">TRUE</property>
            <property name="xalign">0</property>
            <binding name="label">
              <closure type="gchararray" function="get_file_name">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
      </object>
    </property>
  </template>
</interface>
```

factory_grid.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkBox">
        <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
        <property name="spacing">20</property>
        <child>
          <object class="GtkImage">
            <property name="icon-size">GTK_ICON_SIZE_LARGE</property>
            <binding name="gicon">
              <closure type="GIcon" function="get_icon">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
        <child>
          <object class="GtkLabel">
            <property name="hexpand">TRUE</property>
            <property name="xalign">0.5</property>
            <binding name="label">
              <closure type="gchararray" function="get_file_name">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
      </object>
    </property>
  </template>
</interface>
```

The two files above are almost same. The difference is:

- The orientation of the box
- The icon size
- The position of the text of the label

```
$ cd list4; diff factory_list.ui factory_grid.ui
6c6
<         <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
---
>         <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
9a10
>             <property name="icon-size">GTK_ICON_SIZE_LARGE</property>
20c21
<             <property name="xalign">0</property>
---
>             <property name="xalign">0.5</property>
```

Two properties "gicon" (property of GtkImage) and "label" (property of GtkLabel) are in the ui files above. Because GFileInfo doesn't have properties correspond to icon or filename, the factory uses closure tag to bind "gicon" and "label" properties to GFileInfo information. A function `get_icon` gets GIcon from the GFileInfo object. And a function `get_file_name` gets a filename from the GFileInfo object.

```
1  GIcon *
2  get_icon (GtkListItem *item, GFileInfo *info) {
3    GIcon *icon;
4
5     /* g_file_info_get_icon can return NULL */
6    icon = G_IS_FILE_INFO (info) ? g_file_info_get_icon (info) : NULL;
7    return icon ? g_object_ref (icon) : NULL;
8  }
9
10 char *
11 get_file_name (GtkListItem *item, GFileInfo *info) {
12   return G_IS_FILE_INFO (info) ? g_strdup (g_file_info_get_name (info)) : NULL;
13 }
```

One important thing is the ownership of the return values. The return value is owned by the caller. So, `g_obect_ref` or `g_strdup` is necessary.

30.4 An activate signal handler of the button action

An activate signal handler `view_activate` switches the view. It does two things.

- Changes the child widget of GtkScrolledWindow.
- Changes the CSS of buttons to show the current state.

```
1  static void
2  view_activated(GSimpleAction *action, GVariant *parameter) {
3    const char *view = g_variant_get_string (parameter, NULL);
4    const char *other;
5    char *css;
6
7    if (strcmp (view, "list") == 0) {
8      other = "grid";
9      gtk_scrolled_window_set_child (scr, GTK_WIDGET (list));
10   }else {
11     other = "list";
12     gtk_scrolled_window_set_child (scr, GTK_WIDGET (grid));
13   }
14   css = g_strdup_printf ("button#btn%s␣{background:␣silver;}␣button#btn%s␣
       {background:␣white;}", view, other);
15   gtk_css_provider_load_from_data (provider, css, -1);
16   g_free (css);
17   g_action_change_state (G_ACTION (action), parameter);
18 }
```

The second parameter of this handler is the target of the clicked button. Its type is GVariant.

- If `btnlist` has been clicked, then `parameter` is a GVariant of the string "list".
- If `btngrid` has been clicked, then `parameter` is a GVariant of the string "grid".

The third parameter `user_data` points NULL and it is ignored here.

- 3: `g_variant_get_string` gets the string from the GVariant variable.
- 7-13: Sets the child of `scr`. The function `gtk_scrolled_window_set_child` decreases the reference count of the old child by one. And it increases the reference count of the new child by one.
- 14-16: Sets the CSS for the buttons. The background of the clicked button will be silver color and the other button will be white.
- 17: Changes the state of the action.

30.5 Activate signal on GtkListView and GtkGridView

Views (GtkListView and GtkGridView) have an "activate" signal. It is emitted when an item in the view is double clicked or the enter key is pressed. You can do anything you like by connecting the "activate" signal to the handler.

The example `list4` launches `tfe` text file editor if the item of the list is a text file.

```
static void
list_activate (GtkListView *list, int position, gpointer user_data) {
  GFileInfo *info = G_FILE_INFO (g_list_model_get_item (G_LIST_MODEL
      (gtk_list_view_get_model (list)), position));
  launch_tfe_with_file (info);
}

static void
grid_activate (GtkGridView *grid, int position, gpointer user_data) {
  GFileInfo *info = G_FILE_INFO (g_list_model_get_item (G_LIST_MODEL
      (gtk_grid_view_get_model (grid)), position));
  launch_tfe_with_file (info);
}

... ...
... ...

  g_signal_connect (GTK_LIST_VIEW (list), "activate", G_CALLBACK (list_activate),
      NULL);
  g_signal_connect (GTK_GRID_VIEW (grid), "activate", G_CALLBACK (grid_activate),
      NULL);
```

The second parameter of each handler is the position of the item (GFileInfo) of the GListModel. So you can get the item with `g_list_model_get_item` function.

30.5.1 Content type and application launch

The function `launch_tfe_with_file` gets a file from the GFileInfo instance. If the file is a text file, it launches `tfe` with the file.

GFileInfo has information about file type. The file type is like "text/plain", "text/x-csrc" and so on. It is called content type. Content type can be got with `g_file_info_get_content_type` function.

```
1  static void
2  launch_tfe_with_file (GFileInfo *info) {
3    GError *err = NULL;
4    GFile *file;
5    GList *files = NULL;
6    const char *content_type;
7    const char *text_type = "text/";
8    GAppInfo *appinfo;
9    int i;
10
11   if (! info)
```

```
12      return;
13    content_type = g_file_info_get_content_type (info);
14  #ifdef debug
15    g_print ("%s\n", content_type);
16  #endif
17    if (! content_type)
18      return;
19    for (i=0;i<5;++i) { /* compare the first 5 characters */
20      if (content_type[i] != text_type[i])
21        return;
22    }
23    appinfo = g_app_info_create_from_commandline ("tfe", "tfe",
          G_APP_INFO_CREATE_NONE, &err);
24    if (err) {
25      g_printerr ("%s\n", err->message);
26      g_error_free (err);
27      return;
28    }
29    err = NULL;
30    file = g_file_new_for_path (g_file_info_get_name (info));
31    files = g_list_append (files, file);
32    if (! (g_app_info_launch (appinfo, files, NULL, &err))) {
33      g_printerr ("%s\n", err->message);
34      g_error_free (err);
35    }
36    g_list_free_full (files, g_object_unref);
37    g_object_unref (appinfo);
38  }
```

- 13: Gets the content type of the file from GFileInfo.
- 14-16: Prints the content type if "debug" is defined. This is only useful to know a content type of a file. If you don't want this, delete or uncomment the definition `#define debug 1` iat line 6 in the source file.
- 17-22: If no content type or the content type doesn't begin with "text/",the function returns.
- 23: Creates GAppInfo object of `tfe` application. GAppInfo is an interface and the variable `appinfo` points a GDesktopAppInfo instance. GAppInfo is a collection of information of applications.
- 32: Launches the application (`tfe`) with an argument `file`. `g_app_info_launch` has four parameters. The first parameter is GAppInfo object. The second parameter is a list of GFile objects. In this function, only one GFile instance is given to `tfe`, but you can give more arguments. The third parameter is GAppLaunchContext, but this program gives NULL instead. The last parameter is the pointer to the pointer to a GError.
- 36: `g_list_free_full` frees the memories used by the list and items.

If your distribution supports GTK 4, using `g_app_info_launch_default_for_uri` is convenient. The function automatically determines the default application from the file and launches it. For example, if the file is text, then it launches GNOME text editor with the file. Such feature comes from desktop.

30.6 Compilation and execution

The source files are located in src/list4 directory. To compile and execute list4, type as follows.

```
$ cd list4 # or cd src/list4. It depends your current directory.
$ meson setup _build
$ ninja -C _build
$ _build/list4
```

Then a file list appears as a list style. Click on a button on the tool bar so that you can change the style to grid or back to list. Double click "list4.c" item, then `tfe` text editor runs with the argument "list4.c". The following is the screenshot.

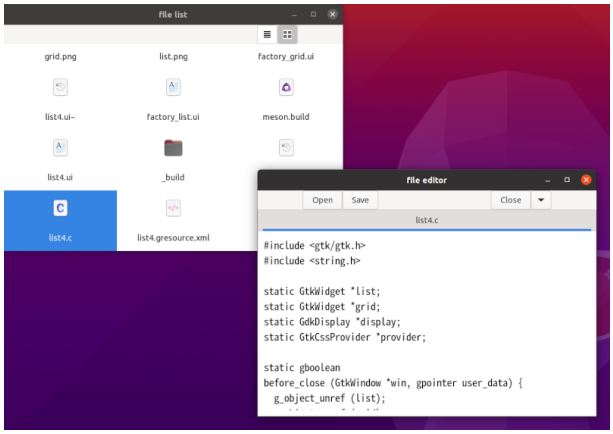


Figure 58: Screenshot

30.7 “gbytes” property of GtkBuilderListItemFactory

GtkBuilderListItemFactory has “gbytes” property. The property contains a byte sequence of ui data. If you use this property, you can put the contents of `factory_list.ui` and `factory_grid.ui`into `list4.ui`. The following shows a part of the new ui file (`list5.ui`).

```
  <object class="GtkListView" id="list">
    <property name="model">
      <object class="GtkSingleSelection" id="singleselection">
        <property name="model">
          <object class="GtkDirectoryList" id="directory_list">
            <property
                name="attributes">standard::name,standard::icon,standard::content-type</property>
          </object>
        </property>
      </object>
    </property>
    <property name="factory">
      <object class="GtkBuilderListItemFactory">
        <property name="bytes"><![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkBox">
        <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
        <property name="spacing">20</property>
        <child>
          <object class="GtkImage">
            <binding name="gicon">
              <closure type="GIcon" function="get_icon">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
        <child>
          <object class="GtkLabel">
            <property name="hexpand">TRUE</property>
            <property name="xalign">0</property>
            <binding name="label">
              <closure type="gchararray" function="get_file_name">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
```

```
        </child>
      </object>
    </property>
  </template>
</interface>
        ]]></property>
      </object>
    </property>
  </object>
```

CDATA section begins with "<![CDATA[" and ends with "]]>". The contents of CDATA section is recognized as a string. Any character, even if it is a key syntax marker such as '<' or '>', is recognized literally. Therefore, the text between "<![CDATA[" and "]]>" is inserted to "bytes" property as it is.

This method decreases the number of ui files. But, the new ui file is a bit complicated especially for the beginners. If you feel some difficulty, it is better for you to separate the ui file.

A directory src/list5 includes the ui file above.

31 GtkExpression

GtkExpression is a fundamental type. It is not a descendant of GObject. GtkExpression provides a way to describe references to values. GtkExpression needs to be evaluated to obtain a value.

It is similar to arithmetic calculation.

```
1 + 2 = 3
```

`1+2` is an expression. It shows the way how to calculate. `3` is the value comes from the expression. Evaluation is to calculate the expression and get the value.

GtkExpression is a way to get a value. Evaluation is like a calculation. A value is got by evaluating the expression.

31.1 Constant expression

A constant expression (GtkConstantExpression) provides constant value or instance when it is evaluated.

```
GValue value = G_VALUE_INIT;
expression = gtk_constant_expression_new (G_TYPE_INT,100);
gtk_expression_evaluate (expression, NULL, &value);
```

- GtkExpression uses GValue to hold a value. GValue is a structure and container to hold a type and value. It must be initialized with `G_VALUE_INIT`, first. Be careful that `value` is a structure, not a pointer to a structure.
- Constant expression is created with `gtk_constant_expression_new` function. The parameter of the function is a type (GType) and a value (or instance). This expression holds a constant value. `G_TYPE_INT` is a type that is registered to the type system. It is integer type. Some types are shown in the following table.
- `gtk_expression_evaluate` evaluates the expression. It has three parameters, the expression to evaluate, `this` instance and a pointer to a GValue for being set with the value. `this` instance isn't necessary for constant expressions. Therefore, the second argument is NULL. `gtk_expression_evaluate` returns TRUE if it successfully evaluates the expression. Otherwise it returns FALSE.
- If it returns TRUE, the GValue `value` is set with the value of the expression. The type of the value is int.

GType	C type	type name	notes
G_TYPE_CHAR	char	gchar	
G_TYPE_BOOLEAN	int	gboolean	
G_TYPE_INT	int	gint	
G_TYPE_FLOAT	float	gfloat	
G_TYPE_DOUBLE	double	gdouble	

GType	C type	type name	notes
G_TYPE_POINTER	void *	gpointer	general pointer
G_TYPE_STRING	char *	gchararray	null-terminated Cstring
G_TYPE_OBJECT		GObject	
GTK_TYPE_WINDOW		GtkWindow	

A sample program `exp_constant_simple.c` is in `src/expression` directory.

```
1  #include <gtk/gtk.h>
2
3  int
4  main (int argc, char **argv) {
5    GtkExpression *expression;
6    GValue value = G_VALUE_INIT;
7
8    /* Create an expression */
9    expression = gtk_constant_expression_new (G_TYPE_INT,100);
10   /* Evaluate the expression */
11   if (gtk_expression_evaluate (expression, NULL, &value))
12     g_print ("The value is %d.\n", g_value_get_int (&value));
13   else
14     g_print ("The constant expression wasn't evaluated correctly.\n");
15   gtk_expression_unref (expression);
16   g_value_unset (&value);
17
18   return 0;
19 }
```

- 9: A constant expression is created. It holds an int value 100. The variable `expression` points the expression.
- 11-14: Evaluates the expression. If it successes, show the value to the stdout. Otherwise show an error message.
- 15-16: Releases the expression and unsets the GValue.

Constant expression is usually used to give a constant value or instance to another expression.

31.2 Property expression

A property expression (GtkPropertyExpression) looks up a property in a GObject instance. For example, a property expression that refers "label" property in a GtkLabel object is created like this.

```
expression = gtk_property_expression_new (GTK_TYPE_LABEL, another_expression,
    "label");
```

The second parameter `another_expression` is one of:

- An expression that gives a GtkLabel instance when it is evaluated.
- NULL. When NULL is given, a GtkLabel instance will be given when it is evaluated. The instance is called `this` object.

For example,

```
label = gtk_label_new ("Hello");
another_expression = gtk_constant_expression_new (GTK_TYPE_LABEL, label);
expression = gtk_property_expression_new (GTK_TYPE_LABEL, another_expression,
    "label");
```

If `expression` is evaluated, the second parameter `another_expression` is evaluated in advance. The value of `another_expression` is the `label` (GtkLabel instance). Then, `expression` looks up "label" property of the label and the evaluation results in "Hello".

In the example above, the second argument of `gtk_property_expression_new` is another expression. But the second argument can be NULL. If it is NULL, `this` instance is used instead. `this` is given by `gtk_expression_evaluate` function.

There's a simple program `exp_property_simple.c` in `src/expression` directory.

```
1  #include <gtk/gtk.h>
2
3  int
4  main (int argc, char **argv) {
5    GtkWidget *label;
6    GtkExpression *expression;
7    GValue value = G_VALUE_INIT;
8
9    gtk_init ();
10   label = gtk_label_new ("Hello␣world.");
11   /* Create an expression */
12   expression = gtk_property_expression_new (GTK_TYPE_LABEL, NULL, "label");
13   /* Evaluate the expression */
14   if (gtk_expression_evaluate (expression, label, &value))
15     g_print ("The␣value␣is␣%s.\n", g_value_get_string (&value));
16   else
17     g_print ("The␣property␣expression␣wasn't␣evaluated␣correctly.\n");
18   gtk_expression_unref (expression);
19   g_value_unset (&value);
20
21   return 0;
22 }
```

- 9-10: `gtk_init` initializes GTK GUI toolkit. It isn't usually necessary because the GtkApplication default startup handler does the initialization. A GtkLabel instance is created with the text "Hello world.".
- 12: A property expression is created. It looks a "label" property of a GtkLabel instance. But at the creation, no instance is given because the second argument is NULL. The expression just knows how to take the property from a future-given GtkLabel instance.
- 14-17: The function `gtk_expression_evaluate` evaluates the expression with a 'this' instance `label`. The result is stored in the GValue `value`. The function `g_value_get_string` gets a string from the GValue. But the string is owned by the GValue so you must not free the string.
- 18-19: Releases the expression and unset the GValue. At the same time the string in the GValue is freed.

If the second argument of `gtk_property_expression_new` isn't NULL, it is another expression. The expression is owned by a newly created property expression. So, when the expressions are useless, you just release the last expression. Then it releases another expression it has.

31.3 Closure expression

A closure expression calls closure when it is evaluated. A closure is a generic representation of a callback (a pointer to a function). For information about closure, see GObject API Reference – The GObject messaging system. There are simple closure example files `closure.c` and `closure_each.c` in the `src/expression` directory.

There are two types of closure expressions, GtkCClosureExpression and GtkClosureExpression. They corresponds to GCClosure and GClosure respectively. When you program in C language, GtkCClosureExpression and GCClosure are appropriate.

A closure expression is created with `gtk_cclosure_expression_new` function.

```
int
callback (GObject *object, int x, const char *s)
```

The following is `exp_closure_simple.c` in `src/expression`.

```
1  #include <gtk/gtk.h>
2
3  static int
4  calc (GtkLabel *label) { /* this object */
5    const char * s;
6    int i, j;
```

```
7
8     s = gtk_label_get_text (label); /* s is owned by the label. */
9     sscanf (s, "%d+%d", &i, &j);
10    return i+j;
11  }
12
13  int
14  main (int argc, char **argv) {
15    GtkExpression *expression;
16    GValue value = G_VALUE_INIT;
17    GtkLabel *label;
18
19    gtk_init ();
20    label = GTK_LABEL (gtk_label_new ("123+456"));
21    g_object_ref_sink (label);
22    expression = gtk_cclosure_expression_new (G_TYPE_INT, NULL, 0, NULL,
23                   G_CALLBACK (calc), NULL, NULL);
24    if (gtk_expression_evaluate (expression, label, &value)) /* 'this' object is the
          label. */
25      g_print ("%d\n", g_value_get_int (&value));
26    else
27      g_print ("The␣closure␣expression␣wasn't␣evaluated␣correctly.\n");
28    gtk_expression_unref (expression);
29    g_value_unset (&value);
30    g_object_unref (label);
31
32    return 0;
33  }
```

- 3-11: A call back function. The parameter is only one and it is a 'this' object. It is a GtkLabel and its label is assumed to be "(number)+(number)".
- 8-10: Retrieves two integers from the label and returns the sum of them. This function has no error report. If you want to return error report, change the return value type to be a pointer to a structure of gboolean and integer. One for error and the other for the sum. The first argument of `gtk_cclosure_expression_new` is `G_TYPE_POINTER`. There is a sample program `exp_closure_with_error_report` in `src/expression` directory.
- 19: The function 'gtk_init" initializes GTK. It is necessary for GtkLabel.
- 20: A GtkLabel instance is created with "123+456".
- 21: The instance has floating reference. It is changed to an ordinary reference count.
- 22-23: Creates a closure expression. Its return value type is `G_TYPE_INT` and no parameters or 'this' object.
- 24: Evaluates the expression with the label as a 'this' object.
- 25: If the evaluation successes, the sum of "123+456", which is 579, is shown.
- 27: If it fails, an error message appears.
- 28-30: Releases the expression and the label. Unsets the value.

Closure expression is flexible than other type of expression because you can specify your own callback function.

31.4 GtkExpressionWatch

GtkExpressionWatch is a structure, not an object. It represents a watched GtkExpression. Two functions create GtkExpressionWatch structure. They are `gtk_expression_bind` and `gtk_expression_watch`.

31.4.1 gtk_expression_bind function

This function binds the target object's property to the expression. If the value of the expression changes, the property reflects the value immediately.

```
GtkExpressionWatch*
gtk_expression_bind (
  GtkExpression* self,
  GObject* target,
```

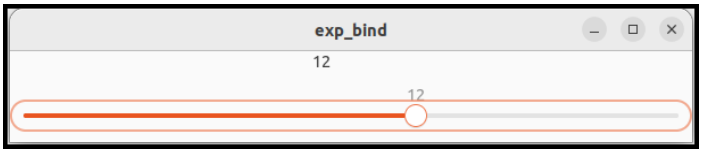


Figure 59: exp_bind

```
  const char* property,
  GObject* this_
)
```

This function takes the ownership of the expression. So, if you want to own the expression, call `gtk_expression_ref ()` to increase the reference count of the expression. And you should unref it when it is useless. If you don't own the expression, you don't care about releasing the expression.

An example `exp_bind.c` and `exp_bind.ui` is in `src/expression` directory.

It includes a label and a scale. If you move the slider to the right, the scale value increases and the number on the label also increases. In the same way, if you move it to the left, the number on the label decreases. The label is bound to the scale value via an adjustment.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <interface>
3    <object class='GtkApplicationWindow' id='win'>
4      <property name='default-width'>600</property>
5      <child>
6        <object class='GtkBox'>
7          <property name='orientation'>GTK_ORIENTATION_VERTICAL</property>
8          <child>
9            <object class='GtkLabel' id='label'>
10             <property name="label">10</property>
11           </object>
12         </child>
13         <child>
14           <object class='GtkScale'>
15             <property name='adjustment'>
16               <object class='GtkAdjustment' id='adjustment'>
17                 <property name='upper'>20.0</property>
18                 <property name='lower'>0.0</property>
19                 <property name='value'>10.0</property>
20                 <property name='step-increment'>1.0</property>
21                 <property name='page-increment'>5.0</property>
22                 <property name='page-size'>0.0</property>
23               </object>
24             </property>
25             <property name='digits'>0</property>
26             <property name='draw-value'>true</property>
27             <property name='has-origin'>true</property>
28             <property name='round-digits'>0</property>
29           </object>
30         </child>
31       </object>
32     </child>
33   </object>
34 </interface>
```

The ui file describes the following parent-child relationship.

```
GtkApplicationWindow (win) -- GtkBox -+- GtkLabel (label)
                                      +- GtkScale
```

Four GtkScale properties are defined.

- adjustment. GtkAdjustment provides the followings.

    - upper and lower: the range of the scale.
    - value: current value of the scale. It reflects the value of the scale.
    - step increment and page increment: When a user press an arrow key or page up/down key, the scale moves by the step increment or page increment respectively.
    - page-size: When an adjustment is used with a scale, page-size is zero.
  - digits: The number of decimal places that are displayed in the value.
  - draw-value: Whether the value is displayed.
  - has-origin: Whether the scale has the origin. If it's true, an orange bar appears between the origin and the current point.
  - round-digits: The number of digits to round the value to when it changes. For example, if it is zero, the slider moves to an integer point.

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

GtkExpressionWatch *watch;

static int
f2i (GObject *object, double d) {
  return (int) d;
}

static int
close_request_cb (GtkWindow *win) {
  gtk_expression_watch_unwatch (watch);
  return false;
}

static void
app_activate (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  gtk_window_present (gtk_application_get_active_window(app));
}

static void
app_startup (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  GtkBuilder *build;
  GtkWidget *win, *label;
  GtkAdjustment *adjustment;
  GtkExpression *expression, *params[1];

  /* Builds a window with exp.ui resource */
  build = gtk_builder_new_from_resource ("/com/github/ToshioCP/exp/exp_bind.ui");
  win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
  label = GTK_WIDGET (gtk_builder_get_object (build, "label"));
  // scale = GTK_WIDGET (gtk_builder_get_object (build, "scale"));
  adjustment = GTK_ADJUSTMENT (gtk_builder_get_object (build, "adjustment"));
  gtk_window_set_application (GTK_WINDOW (win), app);
  g_signal_connect (win, "close-request", G_CALLBACK (close_request_cb), NULL);
  g_object_unref (build);

  /* GtkExpressionWatch */
  params[0] = gtk_property_expression_new (GTK_TYPE_ADJUSTMENT, NULL, "value");
  expression = gtk_cclosure_expression_new (G_TYPE_INT, NULL, 1, params, G_CALLBACK
      (f2i), NULL, NULL);
  watch = gtk_expression_bind (expression, label, "label", adjustment); /* watch
      takes the ownership of the expression. */
}

#define APPLICATION_ID "com.github.ToshioCP.exp_watch"

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

```
51    int stat;
52
53    app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
54
55    g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
56    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
57
58    stat =g_application_run (G_APPLICATION (app), argc, argv);
59    g_object_unref (app);
60    return stat;
61  }
```

The point of the program is:

- 41-42: Two expressions are defined. One is a property expression and the other is a closure expression. The property expression looks up the "value" property of the adjustment instance. The closure expression just converts the double into an integer.
- 43: `gtk_expression_bind` binds the label property of the GtkLabel instance to the integer returned by the closure expression. It creates a GtkExpressionWatch structure. The binding works during the watch lives. When the window is destroyed, the scale and adjustment are also destroyed. And the watch recognizes the value of the expression changes and tries to change the property of the label. Obviously, it is not a correct behavior. The watch should be unwatched before the window is destroyed.
- 37: Connects the "close-request" signal on the window to a handler `close_request_cb`. This signal is emitted when the close button is clicked. The handler is called just before the window closes. It is the right moment to make the GtkExpressionWatch unwatched.
- 10-14: "close-request" signal handler. The function `gtk_expression_watch_unwatch (watch)` makes the watch stop watching the expression. It also releases the expression.

If you want to bind a property to an expression, `gtk_expression_bind` is the best choice. You can do it with `gtk_expression_watch` function, but it is less suitable.

31.4.2 gtk_expression_watch function

```
GtkExpressionWatch*
gtk_expression_watch (
  GtkExpression* self,
  GObject* this_,
  GtkExpressionNotify notify,
  gpointer user_data,
  GDestroyNotify user_destroy
)
```

The function doesn't take the ownership of the expression. It differs from `gtk_expression_bind`. So, you need to release the expression when it is useless. It creates a GtkExpressionWatch structure. The third parameter `notify` is a callback to invoke when the expression changes. You can set `user_data` to give it to the callback. The last parameter is a function to destroy the `user_data` when the watch is unwatched. Put NULL if you don't need them.

Notify callback has the following format.

```
void
notify (
  gpointer user_data
)
```

This function is used to do something when the value of the expression changes. But if you want to bind a property to the value, use `gtk_expression_bind` instead.

There's a sample program `exp_watch.c` in `src/expression` directory. It outputs the width of the window to the standard output.

When you resize the window, the width is displayed in the terminal.

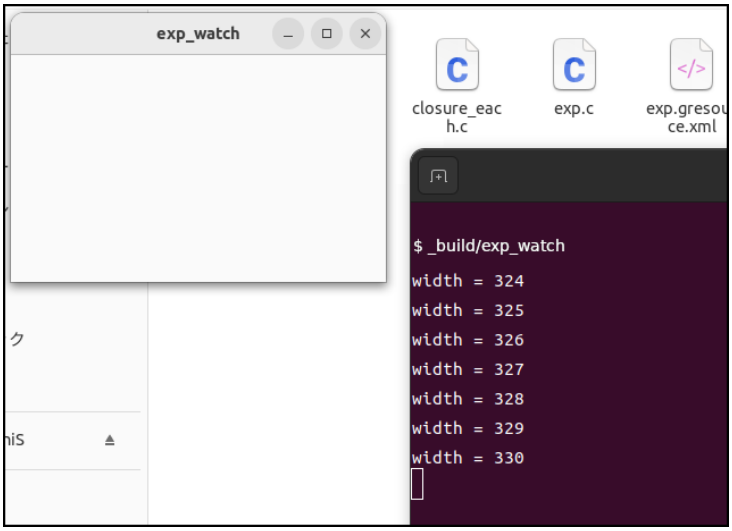


Figure 60: exp_watch

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

GtkExpression *expression;
GtkExpressionWatch *watch;

static void
notify (gpointer user_data) {
  GValue value = G_VALUE_INIT;

  if (gtk_expression_watch_evaluate (watch, &value))
    g_print ("width = %d\n", g_value_get_int (&value));
  else
    g_print ("evaluation failed.\n");
}

static int
close_request_cb (GtkWindow *win) {
  gtk_expression_watch_unwatch (watch);
  gtk_expression_unref (expression);
  return false;
}

static void
app_activate (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  gtk_window_present (gtk_application_get_active_window(app));
}

static void
app_startup (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  GtkWidget *win;

  win = GTK_WIDGET (gtk_application_window_new (app));
  g_signal_connect (win, "close-request", G_CALLBACK (close_request_cb), NULL);

  expression = gtk_property_expression_new (GTK_TYPE_WINDOW, NULL, "default-width");
  watch = gtk_expression_watch (expression, win, notify, NULL, NULL);
}

#define APPLICATION_ID "com.github.ToshioCP.exp_watch"

```

```
43 int
44 main (int argc, char **argv) {
45   GtkApplication *app;
46   int stat;
47
48   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
49
50   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
51   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
52
53   stat =g_application_run (G_APPLICATION (app), argc, argv);
54   g_object_unref (app);
55   return stat;
56 }
```

- 37: A property expression looks up the “default-width” property of the window.
- 38: Create a watch structure for the expression. The callback `notify` is called every time the value of the expression changes. The ‘this’ object is `win`, so the expression returns the default width of the window.
- 6-14: The callback function `notify`. It uses `gtk_expression_watch_evaluate` to get the value of the expression. The ‘this’ object is given in advance (when the watch is created). It outputs the window width to the standard output.
- 16-21: A handler for the “close-request”signal on the window. It stops the watch. In addition, it releases the reference to the expression. Because `gtk_expression_watch` doesn’t take the ownership of the expression, you own it. So, the release is necessary.

31.5 Gtkexpression in ui files

GtkBuilder supports GtkExpressions. There are four tags.

- constant tag to create constant expression. Type attribute specifies the type name of the value. If no type is specified, the type is assumed to be an object. The content is the value of the expression.
- lookup tag to create property expression. Type attribute specifies the type of the object. Name attribute specifies the property name. The content is an expression or object which has the property to look up. If there’s no content, ‘this’ object is used.
- closure tag to create closure expression. Type attribute specifies the type of the returned value. Function attribute specifies the callback function. The contents of the tag are arguments that are expressions.
- binding tag to bind a property to an expression. It is put in the content of an object tag. Name attribute specifies the property name of the object. The content is an expression.

```
<constant type="gchararray">Hello world</constant>
<lookup name="label" type="GtkLabel">label</lookup>
<closure type="gint" function="callback_function"></closure>
<bind name="label">
  <lookup name="default-width">win</lookup>
</bind>
```

These tags are usually used for GtkBuilderListItemFactory.

```
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkLabel">
        <binding name="label">
          <lookup name="string" type="GtkStringObject">
            <lookup name="item">GtkListItem</lookup>
          </lookup>
        </binding>
      </object>
    </property>
  </template>
</interface>
```

Figure 61: exp.c

GtkBuilderListItemFactory uses GtkBuilder to extends the GtkListItem with the XML data.

In the xml file above, "GtkListItem" is an instance of the GtkListItem template. It is the 'this' object given to the expressions. (The information is in the GTK Development Blog).

GtkBuilder calls `gtk_expression_bind` function when it finds a binding tag. The function sets the 'this' object like this:

1. If the binding tag has object attribute, the object will be the 'this' object.
2. If the current object of the GtkBuilder exists, it will be the 'this' object. That's why a GtkListItem instance is the 'this' object of the XML data for a GtkBuilderListItemFactory.
3. Otherwise, the target object of the binding tag will be the 'this' object.

GTK 4 document doesn't describe information about "this" object when expressions are defined in a ui file. The information above is found from the GTK 4 source files and it is possible to include mistakes. If you have accurate information, please let me know.

A sample program `exp.c` and a ui file `exp.ui` is in `src/expression` directory. The ui file includes lookup, closure and bind tags. No constant tag is included. However, constant tags are not used so often.

If you resize the window, the size is shown at the title of the window. If you type characters in the entry, the same characters appear on the label.

The ui file is as follows.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <interface>
3    <object class="GtkApplicationWindow" id="win">
4      <binding name="title">
5        <closure type="gchararray" function="set_title">
6          <lookup name="default-width" type="GtkWindow"></lookup>
7          <lookup name="default-height" type="GtkWindow"></lookup>
8        </closure>
9      </binding>
10     <property name="default-width">600</property>
11     <property name="default-height">400</property>
12     <child>
13       <object class="GtkBox">
14         <property name="orientation">GTK_ORIENTATION_VERTICAL</property>
15         <child>
16           <object class="GtkLabel">
17             <binding name="label">
18               <lookup name="text">
```

```
19                buffer
20              </lookup>
21            </binding>
22          </object>
23        </child>
24        <child>
25          <object class="GtkEntry">
26            <property name="buffer">
27              <object class="GtkEntryBuffer" id="buffer"></object>
28            </property>
29          </object>
30        </child>
31      </object>
32    </child>
33  </object>
34 </interface>
```

- 4-9: The title property of the main window is bound to a closure expression. Its callback function `set_title` is defined in the C source file. It returns a string because the type attribute of the tag is "gchararray". Two parameters are given to the function. They are width and height of the window. Lookup tags don't have contents, so 'this' object is used to look up the properties. The 'this' object is `win`, which is the target of the binding (`win` includes the binding tag).
- 17-21: The "label" property of the GtkLabel instance is bound to the "text" property of `buffer`, which is the buffer of GtkEntry defined in line 25. If a user types characters in the entry, the same characters appear on the label.

The C source file is as follows.

```
1  #include <gtk/gtk.h>
2
3  char *
4  set_title (GtkWidget *win, int width, int height) {
5    return g_strdup_printf ("%d␣x␣%d", width, height);
6  }
7
8  static void
9  app_activate (GApplication *application) {
10   GtkApplication *app = GTK_APPLICATION (application);
11   gtk_window_present (gtk_application_get_active_window(app));
12 }
13
14 static void
15 app_startup (GApplication *application) {
16   GtkApplication *app = GTK_APPLICATION (application);
17   GtkBuilder *build;
18   GtkWidget *win;
19
20   build = gtk_builder_new_from_resource ("/com/github/ToshioCP/exp/exp.ui");
21   win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
22   gtk_window_set_application (GTK_WINDOW (win), app);
23   g_object_unref (build);
24 }
25
26 #define APPLICATION_ID "com.github.ToshioCP.exp"
27
28 int
29 main (int argc, char **argv) {
30   GtkApplication *app;
31   int stat;
32
33   app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
34
35   g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
36   g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
```

```
37
38    stat =g_application_run (G_APPLICATION (app), argc, argv);
39    g_object_unref (app);
40    return stat;
41  }
```

- 4-6: The callback function. It returns a string (w)x(h), where the w and h are the width and height of the window. String duplication is necessary.

The C source file is very simple because almost everything is done in the ui file.

31.6 Conversion between GValues

If you bind different type properties, type conversion is automatically done. Suppose a label property (string) is bound to default-width property (int).

```
<object class="GtkLabel">
  <binding name="label">
    <lookup name="default-width">
      win
    </lookup>
  </binding>
</object>
```

The expression created by the lookup tag returns a int type GValue. On the other hand "label" property holds a string type GValue. When a GValue is copied to another GValue, the type is automatically converted if possible. If the current width is 100, an int `100` is converted to a string `"100"`.

If you use `g_object_get` and `g_object_set` to copy properties, the value is automatically converted.

31.7 Meson.build

The source files are in `src/expression` directory. You can build all the files at once.

```
$ cd src/expression
$ meson setup _build
$ ninja -C _build
```

For example, if you want to run "exp", which is the executable file from "exp.c", type `_build/exp`. You can run other programs as well.

The file `meson.build` is as follows.

```
1  project('exp', 'c')
2
3  gtkdep = dependency('gtk4')
4
5  gnome=import('gnome')
6  resources = gnome.compile_resources('resources','exp.gresource.xml')
7
8  sourcefiles=files('exp.c')
9
10 executable('exp', sourcefiles, resources, dependencies: gtkdep, export_dynamic:
       true, install: false)
11 executable('exp_constant', 'exp_constant.c', dependencies: gtkdep, export_dynamic:
       true, install: false)
12 executable('exp_constant_simple', 'exp_constant_simple.c', dependencies: gtkdep,
       export_dynamic: true, install: false)
13 executable('exp_property_simple', 'exp_property_simple.c', dependencies: gtkdep,
       export_dynamic: true, install: false)
14 executable('closure', 'closure.c', dependencies: gtkdep, export_dynamic: true,
       install: false)
15 executable('closure_each', 'closure_each.c', dependencies: gtkdep, export_dynamic:
       true, install: false)
16 executable('exp_closure_simple', 'exp_closure_simple.c', dependencies: gtkdep,
       export_dynamic: true, install: false)
```

Figure 62: Column View

```
17 executable('exp_closure_with_error_report', 'exp_closure_with_error_report.c',
       dependencies: gtkdep, export_dynamic: true, install: false)
18 executable('exp_bind', 'exp_bind.c', resources, dependencies: gtkdep,
       export_dynamic: true, install: false)
19 executable('exp_watch', 'exp_watch.c', dependencies: gtkdep, export_dynamic: true,
       install: false)
20 executable('exp_test', 'exp_test.c', resources, dependencies: gtkdep,
       export_dynamic: true, install: false)
```

32 GtkColumnView

32.1 GtkColumnView

GtkColumnView is like GtkListView, but it has multiple columns. Each column is GtkColumnViewColumn.

- GtkColumnView has “model” property. The property points a GtkSelectionModel object.
- Each GtkColumnViewColumn has “factory” property. The property points a GtkListItemFactory (GtkSignalListItemFactory or GtkBuilderListItemFactory).
- The factory connects GtkListItem and items of GtkSelectionModel. And the factory builds the descendant widgets of GtkColumnView to display the item on the display. This process is the same as the one in GtkListView.

The following diagram shows how it works.

The example in this section is a window that displays information of files in a current directory. The information is the name, size and last modified datetime of files. So, there are three columns.

In addition, the example uses GtkSortListModel and GtkSorter to sort the information.

32.2 column.ui

Ui file specifies widgets and list item templates.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <interface>
3   <object class="GtkApplicationWindow" id="win">
4     <property name="title">file list</property>
```

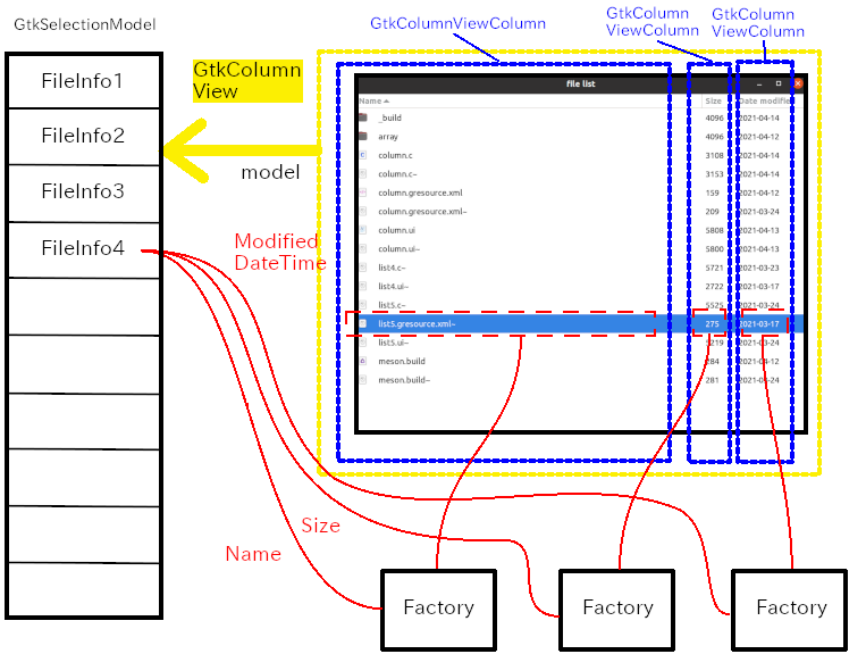


Figure 63: ColumnView

```
5       <property name="default-width">800</property>
6       <property name="default-height">600</property>
7       <child>
8         <object class="GtkScrolledWindow">
9           <property name="hexpand">TRUE</property>
10          <property name="vexpand">TRUE</property>
11          <child>
12            <object class="GtkColumnView" id="columnview">
13              <property name="model">
14                <object class="GtkNoSelection">
15                  <property name="model">
16                    <object class="GtkSortListModel">
17                      <property name="model">
18                        <object class="GtkDirectoryList" id="directorylist">
19                          <property
                                name="attributes">standard::name,standard::icon,standard::size,time::m
20                        </object>
21                      </property>
22                      <binding name="sorter">
23                        <lookup name="sorter">columnview</lookup>
24                      </binding>
25                    </object>
26                  </property>
27                </object>
28              </property>
29              <child>
30                <object class="GtkColumnViewColumn">
31                  <property name="title">Name</property>
32                  <property name="expand">TRUE</property>
33                  <property name="factory">
34                    <object class="GtkBuilderListItemFactory">
35                      <property name="bytes"><![CDATA[
36  <?xml version="1.0" encoding="UTF-8"?>
37  <interface>
38    <template class="GtkListItem">
39      <property name="child">
40        <object class="GtkBox">
```

```
        <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
        <property name="spacing">20</property>
        <child>
          <object class="GtkImage">
            <binding name="gicon">
              <closure type="GIcon" function="get_icon_factory">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
        <child>
          <object class="GtkLabel">
            <property name="hexpand">TRUE</property>
            <property name="xalign">0</property>
            <binding name="label">
              <closure type="gchararray" function="get_file_name_factory">
                <lookup name="item">GtkListItem</lookup>
              </closure>
            </binding>
          </object>
        </child>
      </object>
    </property>
  </template>
</interface>
                    ]]></property>
                  </object>
                </property>
                <property name="sorter">
                  <object class="GtkStringSorter">
                    <property name="expression">
                      <closure type="gchararray" function="get_file_name">
                      </closure>
                    </property>
                  </object>
                </property>
              </object>
            </child>
            <child>
              <object class="GtkColumnViewColumn">
                <property name="title">Size</property>
                <property name="factory">
                  <object class="GtkBuilderListItemFactory">
                    <property name="bytes"><![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkLabel">
        <property name="hexpand">TRUE</property>
        <property name="xalign">0</property>
        <binding name="label">
          <closure type="gint64" function="get_file_size_factory">
            <lookup name="item">GtkListItem</lookup>
          </closure>
        </binding>
      </object>
    </property>
  </template>
</interface>
                    ]]></property>
                  </object>
                </property>
```

```
                <property name="sorter">
                  <object class="GtkNumericSorter">
                    <property name="expression">
                      <closure type="gint64" function="get_file_size">
                      </closure>
                    </property>
                    <property name="sort-order">GTK_SORT_ASCENDING</property>
                  </object>
                </property>
              </object>
            </child>
            <child>
              <object class="GtkColumnViewColumn">
                <property name="title">Date modified</property>
                <property name="factory">
                  <object class="GtkBuilderListItemFactory">
                    <property name="bytes"><![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkLabel">
        <property name="hexpand">TRUE</property>
        <property name="xalign">0</property>
        <binding name="label">
          <closure type="gchararray" function="get_file_time_modified_factory">
            <lookup name="item">GtkListItem</lookup>
          </closure>
        </binding>
      </object>
    </property>
  </template>
</interface>
                    ]]></property>
                  </object>
                </property>
                <property name="sorter">
                  <object class="GtkNumericSorter">
                    <property name="expression">
                      <closure type="gint64" function="get_file_unixtime_modified">
                      </closure>
                    </property>
                    <property name="sort-order">GTK_SORT_ASCENDING</property>
                  </object>
                </property>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

- 3-12: GtkApplicationWindow has a child widget GtkScrolledWindow. GtkScrolledWindoww has a child widget GtkColumnView.
- 12-18: GtkColumnView has “model” property. It points GtkSelectionModel interface. GtkNoSelection class is used as GtkSelectionModel. And again, it has “model” property. It points GtkSortListModel. This list model supports sorting the list. It will be explained in the later subsection. And it also has “model” property. It points GtkDirectoryList. Therefore, the chain is: GtkColumnView => GtkNoSelection => GtkSortListModel => GtkDirectoryList.
- 18-20: GtkDirectoryList. It is a list of GFileInfo, which holds information of files under a directory. It has “attributes” property. It specifies what attributes is kept in each GFileInfo.

  - "standard::name" is a name of the file.
  - "standard::icon" is a GIcon object of the file
  - "standard::size" is the file size.
  - "time::modified" is the date and time the file was last modified.
- 29-79: The first GtkColumnViewColumn object. There are four properties, "title", "expand", factory" and "sorter".
- 31: Sets the "title" property to "Name". This is the title on the header of the column.
- 32: Sets the "expand" property to TRUE to allow the column to expand as much as possible. (See the image above).
- 33- 69: Sets the "factory" property to GtkBuilderListItemFactory. The factory has "bytes" property which holds a ui string to define a template to extend GtkListItem class. The CDATA section (line 36-66) is the ui string to put into the "bytes" property. The contents are the same as the ui file `factory_list.ui` in the section 30.
- 70-77: Sets the "sorter" property to GtkStringSorter object. This object provides a sorter that compares strings. It has "expression" property. A closure tag with a string type function `get_file_name` is used here. The function will be explained later.
- 80-115: The second GtkColumnViewColumn object. Its sorter property is set to GtkNumericSorter.
- 116-151: The third GtkColumnViewColumn object. Its sorter property is set to GtkNumericSorter.

32.3 GtkSortListModel and GtkSorter

GtkSortListModel is a list model that sorts its elements according to a GtkSorter instance assigned to the "sorter" property. The property is bound to "sorter" property of GtkColumnView in line 22 to 24.

```
<object class="GtkSortListModel" id="sortlist">
... ... ...
  <binding name="sorter">
    <lookup name="sorter">columnview</lookup>
  </binding>
```

Therefore, `columnview` determines the way how to sort the list model. The "sorter" property of GtkColumnView is read-only property and it is a special sorter. It reflects the user's sorting choice. If a user clicks the header of a column, then the sorter ("sorter" property) of the column is referenced by "sorter" property of the GtkColumnView. If the user clicks the header of another column, then the "sorter" property of the GtkColumnView refers to the newly clicked column's "sorter" property.

The binding above makes a indirect connection between the "sorter" property of GtkSortListModel and the "sorter" property of each column.

GtkSorter compares two items (GObject or its descendant). GtkSorter has several child objects.

- GtkStringSorter compares strings taken from the items.
- GtkNumericSorter compares numbers taken from the items.
- GtkCustomSorter uses a callback to compare.
- GtkMultiSorter combines multiple sorters.

The example uses GtkStringSorter and GtkNumericSorter.

GtkStringSorter uses GtkExpression to get the strings from the items (objects). The GtkExpression is stored in the "expression" property of the GtkStringSorter. When GtkStringSorter compares two items, it evaluates the expression by calling `gtk_expression_evaluate` function. It assigns each item to the second argument ('this' object) of the function.

In the ui file above, the GtkExpression is in the line 71 to 76.

```
<object class="GtkStringSorter">
  <property name="expression">
    <closure type="gchararray" function="get_file_name">
    </closure>
  </property>
</object>
```

The GtkExpression calls `get_file_name` function when it is evaluated.

```
char *
get_file_name (GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_strdup(g_file_info_get_name (info)) : NULL;
}
```

The function is given the item (GFileInfo) of the GtkSortListModel as an argument (`this` object). But you need to be careful that it can be NULL while the list item is being recycled. So, `G_IS_FILE_INFO (info)` is always necessary in callback functions. The function retrieves a filename from `info`. The string is owned by `info` so it is necessary to duplicate. And it returns the copied string.

GtkNumericSorter compares numbers. It is used in the line 106 to 112 and line 142 to 148. The lines from 106 to 112 is:

```
<object class="GtkNumericSorter">
  <property name="expression">
    <closure type="gint64" function="get_file_size">
    </closure>
  </property>
  <property name="sort-order">GTK_SORT_ASCENDING</property>
</object>
```

The closure tag specifies a callback function `get_file_size`.

```
goffset
get_file_size (GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_file_info_get_size (info): -1;
}
```

It just returns the size of `info`. The type of the size is `goffset`. The type `goffset` is the same as `gint64`.

The lines from 142 to 148 is:

```
<object class="GtkNumericSorter" id="sorter_datetime_modified">
  <property name="expression">
    <closure type="gint64" function="get_file_unixtime_modified">
    </closure>
  </property>
  <property name="sort-order">GTK_SORT_ASCENDING</property>
</object>
```

The closure tag specifies a callback function `get_file_unixtime_modified`.

```
gint64
get_file_unixtime_modified (GFileInfo *info) {
  GDateTime *dt;

  dt = G_IS_FILE_INFO (info) ? g_file_info_get_modification_date_time (info) : NULL;
  return dt ? g_date_time_to_unix (dt) : -1;
}
```

It gets the modification date and time (GDateTime type) of `info`. Then it gets a unix time from `dt`. Unix time, sometimes called unix epoch, is the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. It returns the unix time (gint64 type).

32.4 column.c

`column.c` is as follows. It is simple and short thanks to `column.ui`.

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

/* functions (closures) for GtkBuilderListItemFactory */
GIcon *
get_icon_factory (GtkListItem *item, GFileInfo *info) {
  GIcon *icon;

   /* g_file_info_get_icon can return NULL */
```

```
  icon = G_IS_FILE_INFO (info) ? g_file_info_get_icon (info) : NULL;
  return icon ? g_object_ref (icon) : NULL;
}

char *
get_file_name_factory (GtkListItem *item, GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_strdup (g_file_info_get_name (info)) : NULL;
}

/* goffset is defined as gint64 */
/* It is used for file offsets. */
goffset
get_file_size_factory (GtkListItem *item, GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_file_info_get_size (info) : -1;
}

char *
get_file_time_modified_factory (GtkListItem *item, GFileInfo *info) {
  GDateTime *dt;

   /* g_file_info_get_modification_date_time can return NULL */
  dt = G_IS_FILE_INFO (info) ? g_file_info_get_modification_date_time (info) : NULL;
  return dt ? g_date_time_format (dt, "%F") : NULL;
}

/* Functions (closures) for GtkSorter */
char *
get_file_name (GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_strdup(g_file_info_get_name (info)) : NULL;
}

goffset
get_file_size (GFileInfo *info) {
  return G_IS_FILE_INFO (info) ? g_file_info_get_size (info): -1;
}

gint64
get_file_unixtime_modified (GFileInfo *info) {
  GDateTime *dt;

  dt = G_IS_FILE_INFO (info) ? g_file_info_get_modification_date_time (info) : NULL;
  return dt ? g_date_time_to_unix (dt) : -1;
}

static void
app_activate (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  gtk_window_present (gtk_application_get_active_window(app));
}

static void
app_startup (GApplication *application) {
  GtkApplication *app = GTK_APPLICATION (application);
  GFile *file;
  GtkBuilder *build = gtk_builder_new_from_resource
      ("/com/github/ToshioCP/column/column.ui");
  GtkWidget *win = GTK_WIDGET (gtk_builder_get_object (build, "win"));
  GtkDirectoryList *directorylist = GTK_DIRECTORY_LIST (gtk_builder_get_object
      (build, "directorylist"));
  g_object_unref (build);

  gtk_window_set_application (GTK_WINDOW (win), app);

  file = g_file_new_for_path (".");
```

Figure 64: Column View

```
71    gtk_directory_list_set_file (directorylist, file);
72    g_object_unref (file);
73  }
74
75  #define APPLICATION_ID "com.github.ToshioCP.columnview"
76
77  int
78  main (int argc, char **argv) {
79    GtkApplication *app;
80    int stat;
81
82    app = gtk_application_new (APPLICATION_ID, G_APPLICATION_DEFAULT_FLAGS);
83
84    g_signal_connect (app, "startup", G_CALLBACK (app_startup), NULL);
85    g_signal_connect (app, "activate", G_CALLBACK (app_activate), NULL);
86
87    stat =g_application_run (G_APPLICATION (app), argc, argv);
88    g_object_unref (app);
89    return stat;
90  }
```

32.5 Compilation and execution.

All the source files are in `src/column` directory. Change your current directory to the directory and type the following.

```
$ cd src/colomn
$ meson setup _build
$ ninja -C _build
$ _build/column
```

Then, a window appears.

If you click the header of a column, then the whole lists are sorted by the column. If you click the header of another column, then the whole lists are sorted by the newly selected column.

Figure 65: List editor

GtkColumnView is very useful and it can manage very big GListModel. It is possible to use it for file list, application list, database frontend and so on.

33 GtkSignalListItemFactory

33.1 GtkSignalListItemFactory and GtkBulderListItemFactory

GtkBuilderlistItemFactory is convenient when GtkListView just shows the contents of a list. Its binding direction is always from an item of a list to a child of GtkListItem.

When it comes to dynamic connection, it's not enough. For example, suppose you want to edit the contents of a list. You set a child of GtkListItem to a GtkText instance so a user can edit a text with it. You need to bind an item in the list with the buffer of the GtkText. The direction is opposite from the one with GtkBuilderListItemFactory. It is from the GtkText instance to the item in the list. You can implement this with GtkSignalListItemFactory, which is more flexible than GtkBuilderListItemFactory.

This section shows just some parts of the source file `listeditor.c`. If you want to see the whole codes, see `src/listeditor` directory of the Gtk4 tutorial repository.

33.2 A list editor

The sample program is a list editor and data of the list are strings. It's the same as a line editor. It reads a text file line by line. Each line is an item of the list. The list is displayed with GtkColumnView. There are two columns. The one is a button, which shows if the line is a current line. If the line is the current line, the button is colored with red. The other is a string which is the contents of the corresponding item of the list.

The source files are located at `src/listeditor` directory. You can compile end execute it as follows.

- Download the program from the repository.
- Change your current directory to `src/listeditor`.
- Type the following on your commandline.

```
$ meson setup _build
$ ninja -C _build
$ _build/listeditor
```

- Append button: appends a line after the current line, or at the last line if no current line exists.

- Insert button: inserts a line before the current line, or at the top line if no current line exists.
- Remove button: removes a current line.
- Read button: reads a file.
- Write button: writes the contents to a file.
- close button: closes the contents.
- quit button: quits the application.
- Button on the select column: makes the line current.
- String column: GtkText. You can edit a string in the field.

The current line number (zero-based) is shown at the left of the tool bar. The file name is shown at the right of the write button.

33.3 Connect a GtkText instance and an item in the list

The second column (GtkColumnViewColumn) sets its factory property to GtkSignalListItemFactory. It uses three signals setup, bind and unbind. The following shows the signal handlers.

```
1  static void
2  setup2_cb (GtkListItemFactory *factory, GtkListItem *listitem) {
3    GtkWidget *text = gtk_text_new ();
4    gtk_list_item_set_child (listitem, GTK_WIDGET (text));
5    gtk_editable_set_alignment (GTK_EDITABLE (text), 0.0);
6  }
7
8  static void
9  bind2_cb (GtkListItemFactory *factory, GtkListItem *listitem) {
10   GtkWidget *text = gtk_list_item_get_child (listitem);
11   GtkEntryBuffer *buffer = gtk_text_get_buffer (GTK_TEXT (text));
12   LeData *data = LE_DATA (gtk_list_item_get_item(listitem));
13   GBinding *bind;
14
15   gtk_editable_set_text (GTK_EDITABLE (text), le_data_look_string (data));
16   gtk_editable_set_position (GTK_EDITABLE (text), 0);
17
18   bind = g_object_bind_property (buffer, "text", data, "string", G_BINDING_DEFAULT);
19   g_object_set_data (G_OBJECT (listitem), "bind", bind);
20 }
21
22 static void
23 unbind2_cb (GtkListItemFactory *factory, GtkListItem *listitem) {
24   GBinding *bind = G_BINDING (g_object_get_data (G_OBJECT (listitem), "bind"));
25
26   if (bind)
27     g_binding_unbind(bind);
28   g_object_set_data (G_OBJECT (listitem), "bind", NULL);
29 }
```

- 1-6: `setup2_cb` is a setup signal handler on the GtkSignalListItemFactory. This factory is inserted to the factory property of the second GtkColumnViewColumn. The handler just creates a GtkText instance and sets the child of `listitem` to it. The instance will be destroyed automatically when the `listitem` is destroyed. So, teardown signal handler isn't necessary.
- 8-20: `bind2_cb` is a bind signal handler. It is called when the `listitem` is bound to an item in the list. The list items are LeData instances. LeData is defined in the file `listeditor.c` (the C source file of the list editor). It is a child class of GObject and has string data which is the content of the line.
  - 10-11: `text` is a child of the `listitem` and it is a GtkText instance. And `buffer` is a GtkEntryBuffer instance of the `text`.
  - 12: The LeData instance `data` is an item pointed by the `listitem`.
  - 15-16: Sets the text of `text` to `le_data_look_string (data)`. le_data_look_string returns the string of the `data` and the ownership of the string is still taken by the `data`. So, the caller doesn't need to free the string.
  - 18: `g_object_bind_property` binds a property and another object property. This line binds the "text" property of the `buffer` (source) and the "string" property of the `data` (destination). It is a uni-directional binding (`G_BINDING_DEFAULT`). When a user changes the GtkText text, the

same string is immediately put into the `data`. The function returns a GBinding instance. This binding is different from bindings of GtkExpression. This binding needs the existence of the two properties.
    - 19: GObjec has a table. The key is a string (or GQuark) and the value is a gpointer (pointer to any type). The function `g_object_set_data` sets the association from the key to the value. This line sets the association from "bind" to `bind` instance. It makes possible for the "unbind" handler to get the `bind` instance.
- 22-29: `unbind2_cb` is a unbind signal handler.
    - 24: Retrieves the `bind` instance from the table in the `listitem` instance.
    - 26-27: Unbind the binding.
    - 28: Removes the value corresponds to the "bind" key.

This technique is not so complicated. You can use it when you make a cell editable application.

If it is impossible to use `g_object_bind_property`, use a notify signal on the GtkEntryBuffer instance. You can use "deleted-text" and "inserted-text" signal instead. The handler of the signals above copies the text in the GtkEntryBuffer instance to the LeData string. Connect the notify signal handler in `bind2_cb` and disconnect it in `unbind2_cb`.

33.4 Change the cell of GtkColumnView dynamically

Next topic is to change the GtkColumnView (or GtkListView) cells dynamically. The example changes the color of the buttons, which are children of GtkListItem instances, as the current line position moves.

The line editor has the current position of the list.

- At first, no line is current.
- When a line is appended or inserted, the line is current.
- When the current line is deleted, no line will be current.
- When a button in the first column of GtkColumnView is clicked, the line will be current.
- It is necessary to set the line status (whether current or not) when a GtkListItem is bound to an item in the list. It is because GtkListItem is recycled. A GtkListItem was possibly current line before but not current after recycled. The opposite can also be happen.

The button of the current line is colored with red and otherwise white.

The current line has no relationship to GtkSingleSelection object. GtkSingleSelection selects a line on the display. The current line doesn't need to be on the display. It is possible to be on the line out of the Window (GtkScrolledWindow). Actually, the program doesn't use GtkSingleSelection.

The LeWindow instance has two instance variables for recording the current line.

- `win->position`: An int type variable. It is the position of the current line. It is zero-based. If no current line exists, it is -1.
- `win->current_button`: A variable points the button, located at the first column, on the current line. If no current line exists, it is NULL.

If the current line moves, the following two functions are called. They updates the two varables.

```
1  static void
2  update_current_position (LeWindow *win, int new) {
3    char *s;
4
5    win->position = new;
6    if (win->position >= 0)
7      s = g_strdup_printf ("%d", win->position);
8    else
9      s = "";
10   gtk_label_set_text (GTK_LABEL (win->position_label), s);
11   if (*s) // s isn't an empty string
12     g_free (s);
13 }
14
15 static void
16 update_current_button (LeWindow *win, GtkButton *new_button) {
```

```
17    const char *non_current[1] = {NULL};
18    const char *current[2] = {"current", NULL};
19
20    if (win->current_button) {
21      gtk_widget_set_css_classes (GTK_WIDGET (win->current_button), non_current);
22      g_object_unref (win->current_button);
23    }
24    win->current_button = new_button;
25    if (win->current_button) {
26      g_object_ref (win->current_button);
27      gtk_widget_set_css_classes (GTK_WIDGET (win->current_button), current);
28    }
29  }
```

The varable `win->position_label` points a GtkLabel instance. The label shows the current line position.

The current button has CSS "current" class. The button is colored red through the CSS "button.current {background: red;}".

The order of the call for these two functions is important. The first function, which updates the position, is usually called first. After that, a new line is appended or inserted. Then, the second function is called.

The following functions call the two functions above. Be careful about the order of the call.

```
1   void
2   select_cb (GtkButton *btn, GtkListItem *listitem) {
3     LeWindow *win = LE_WINDOW (gtk_widget_get_ancestor (GTK_WIDGET (btn),
          LE_TYPE_WINDOW));
4
5     update_current_position (win, gtk_list_item_get_position (listitem));
6     update_current_button (win, btn);
7   }
8
9   static void
10  setup1_cb (GtkListItemFactory *factory, GtkListItem *listitem) {
11    GtkWidget *button = gtk_button_new ();
12    gtk_list_item_set_child (listitem, button);
13    gtk_widget_set_focusable (GTK_WIDGET (button), FALSE);
14    g_signal_connect (button, "clicked", G_CALLBACK (select_cb), listitem);
15  }
16
17  static void
18  bind1_cb (GtkListItemFactory *factory, GtkListItem *listitem, gpointer user_data) {
19    LeWindow *win = LE_WINDOW (user_data);
20    GtkWidget *button = gtk_list_item_get_child (listitem);
21
22    if (win->position == gtk_list_item_get_position (listitem))
23      update_current_button (win, GTK_BUTTON (button));
24  }
```

- 1-7: `select_cb` is a "clicked" signal handler. The handler just calls the two functions and update the position and button.
- 9-15: `setup1_cb` is a setup signal handler on the GtkSignalListItemFactory. It sets the child of `listitem` to a GtkButton instance. The "clicked" signal on the button is connected to the handler `select_cb`. When the listitem is destroyed, the child (GtkButton) is also destroyed. At the same time, the connection of the signal and the handler is also destroyed. So, you don't need teardown signal handler.
- 17-24: `bind1_cb` is a bind signal handler. Usually, the position moves before this handler is called. If the item is on the current line, the button is updated. No unbind handler is necessary.

When a line is added, the current position is updated in advance.

```
1   static void
2   app_cb (GtkButton *btn, LeWindow *win) {
3     LeData *data = le_data_new_with_data ("");
```

```
4
5     if (win->position >= 0) {
6       update_current_position (win, win->position + 1);
7       g_list_store_insert (win->liststore, win->position, data);
8     } else {
9       update_current_position (win, g_list_model_get_n_items (G_LIST_MODEL
            (win->liststore)));
10      g_list_store_append (win->liststore, data);
11    }
12    g_object_unref (data);
13  }
14
15  static void
16  ins_cb (GtkButton *btn, LeWindow *win) {
17    LeData *data = le_data_new_with_data ("");
18
19    if (win->position >= 0)
20      g_list_store_insert (win->liststore, win->position, data);
21    else {
22      update_current_position (win, 0);
23      g_list_store_insert (win->liststore, 0, data);
24    }
25    g_object_unref (data);
26  }
```

When a line is removed, the current position becomes -1 and no button is current.

```
1   static void
2   rm_cb (GtkButton *btn, LeWindow *win) {
3     if (win->position >= 0) {
4       g_list_store_remove (win->liststore, win->position);
5       update_current_position (win, -1);
6       update_current_button (win, NULL);
7     }
8   }
```

The color of buttons are determined by the "background" CSS style. The following CSS node is a bit complicated. CSS node `column view` has `listview` child node. It covers the rows in the GtkColumnView. The `listview` node is the same as the one for GtkListView. It has `row` child node, which is for each child widget. Therefore, the following node corresponds buttons on the GtkColumnView widget. In addition, it is applied to the "current" class.

```
columnview listview row button.current {background: red;}
```

33.5 A waring from GtkText

If your program has the following two, a warning message can be issued.

- The list has many items and it needs to be scrolled.
- A GtkText instance is the focus widget.

```
GtkText - unexpected blinking selection. Removing
```

I don't have an exact idea why this happens. But if GtkText "focusable" property is FALSE, the warning doesn't happen. So it probably comes from focus and scroll.

You can avoid this by unsetting any focus widget under the main window. When scroll begins, the "value-changed" signal on the vertical adjustment of the scrolled window is emitted.

The following is extracted from the ui file and C source file.

```
... ... ...
<object class="GtkScrolledWindow">
  <property name="hexpand">TRUE</property>
  <property name="vexpand">TRUE</property>
```

```
    <property name="vadjustment">
      <object class="GtkAdjustment">
        <signal name="value-changed" handler="adjustment_value_changed_cb"
            swapped="no" object="LeWindow"/>
      </object>
    </property>
  ... ... ...
```

```
1 static void
2 adjustment_value_changed_cb (GtkAdjustment *adjustment, gpointer user_data) {
3   GtkWidget *win = GTK_WIDGET (user_data);
4
5   gtk_window_set_focus (GTK_WINDOW (win), NULL);
6 }
```

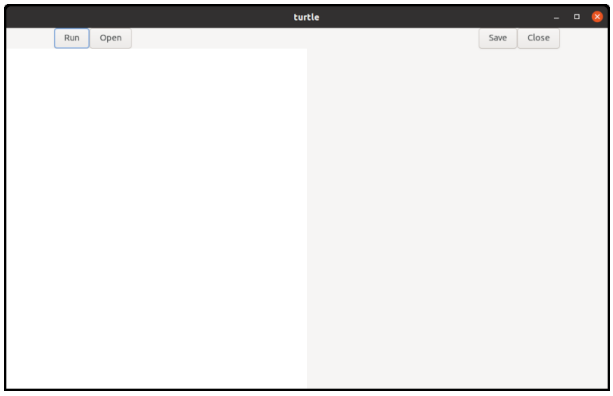


Figure 66: Screenshot just after it's executed

A Turtle manual

Turtle is a simple interpreter for turtle graphics.

A.1 Prerequisite and compiling

Turtle is written in C language. You need:

- Linux. Turtle is tested on ubuntu 22.10
- gcc, meson and ninja
- gtk4

It is easy to compile the source file of turtle. If you want tp install it in your local area, put an option `--prefix=$HOME/.local` to your meson command line. Then, it will be installed under `$HOME/.local/bin`. The instruction is:

```
$ meson setup --prefix=$HOME/.local _build
$ ninja -C _build
$ ninja -C _build install
```

If you want to install it in the system area, no option is necessary. It will be installed under `/usr/local/bin`.

```
$ meson setup _build
$ ninja -C _build
$ sudo ninja -C _build install
```

Type the following command then turtle shows the following window.

```
$ turtle
```

The left half is a text editor and the right half is a surface. Surface is like a canvas to draw shapes.

Write turtle language in the text editor and click on `run` button, then the program will be executed and it draws shapes on the surface.

If you uncomment the following line in `turtle.y`, then codes for debug will also be compiled. Turtle shows the status to the standard output, but the speed is quite slow. It is not recommended except you are developing the program.

```
/* # define debug 1 */
```

A.2 Example

Imagine a turtle. The turtle has a pen and initially it is at the center of the screen, facing to the north (to the north means up on the screen). You can let the turtle down the pen or up the pen. You can order the turtle to move forward.

```
pd
fd 100
```

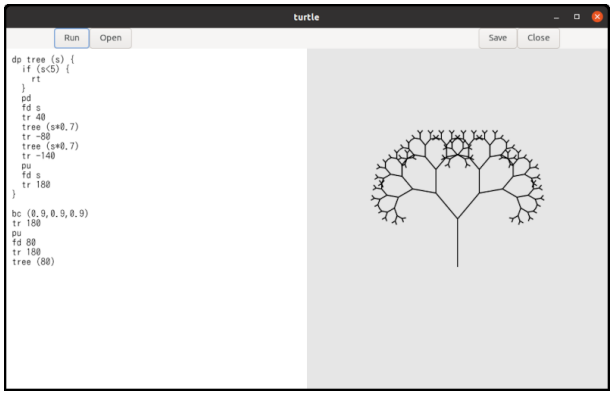


Figure 67: Tree

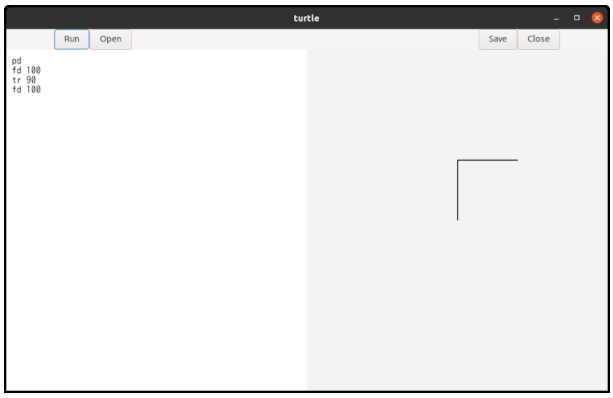


Figure 68: Two line segments on the surface

- pd: Pen Down. The turtle put the pen down so that the turtle will draw a line if it moves.
- fd 100: move ForwarD 100. The turtle goes forward 100 pixels.

If you click on `run` button, then a line segment appears on the screen. One of the endpoints of the line segment is at the center of the surface and the other is at 100 pixels up from the center. The point at the center is the start point of the turtle and the other endpoint is the end point of the movement.

If the turtle picks the pen up, then no line segment appears.

```
pu
fd 100
```

The command `pu` means "Pen Up".

The turtle can change the direction.

```
pd
fd 100
tr 90
fd 100
```

The command `tr` is "Turn Right". The argument is angle with degrees. Therefore, `tr 90` means "Turn right by 90 degrees". If you click on the `run`button, then two line segments appears. One is vertical and the other is horizontal.

You can use `tl` (Turn Left) as well.

A.3 Background and foreground color

Colors are specified with RGB. A vector (r, g, b) denotes RGB color. Each of the elements is a real number between 0 and 1.

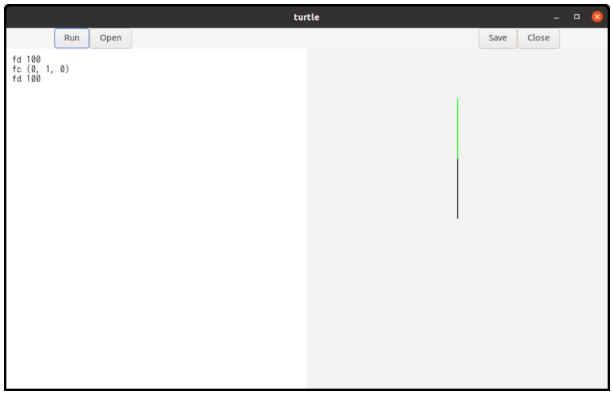


Figure 69: Change the foreground color

- Red is (1.0, 0.0, 0.0). You can write (1, 0, 0) instead.
- Green is (0.0, 1.0, 0.0)
- Blue is (0.0, 0.0, 1.0)
- Black is (0.0, 0.0, 0.0)
- White is (1.0, 1.0, 1.0)

You can express a variety of colors by changing each element.

There are two commands to change colors.

- bc: Background Color. `bc (1,0,0)` changes the background color to red. This command clear the surface and change the background color. So, the shapes on the surface disappears.
- fc: Foreground Color. `fc (0,1,0)` changes the foreground color to green. This command changes the pen color. The prior shapes on the surface aren't affected. After this command, the turtle draws lines with the new color.

A.4 Other simple commands

- pw: Pen Width. This is the same as pen size or line width. For example, `pw 5` makes lines thick and `pw 1` makes it thin.
- rs: ReSet. The turtle moves back to the initial position and direction. In addition, The command initialize the pen, line width (pen size), and foreground color. The pen is down, the line width is 2 and the foreground color is black.

An order such as `fd 100`, `pd` and so on is a statement. Statements are executed in the order from the top to the end in the program.

A.5 Comment and spaces

Characters between `#` (hash mark) and `\n` (new line) are comment. If the comment is at the end of the file, the trailing new line can be left out. Comments are ignored.

```
# draw a triangle<NEW LINE>
fd 100 # forward 100 pixels<NEW LINE>
tr 120 # turn right by 90 degrees<NEW LINE>
fd 100<NEW LINE>
tr 120<NEW LINE>
fd 100 # Now a triangle appears.<EOF>
```

<NEW LINE> and <EOF> indicate newline code and end of file respectively. The comments in the line 1, 2, 3 and 6 are correct syntactically.

Spaces (white space, tab and new line) are ignored. They are used only as delimiters. Tabs are recognized as eight spaces to calculate the column number.

A.6 Variables and expressions

Variables begin alphabet followed by alphabet or digit. Key words like `fd` or `tr` can't be variables. `Distance` and `angle5` are variables, but `1step` isn't a variable because the first character isn't alphabet. Variable names are case sensitive. Variables keep real numbers. Their type is the same as `double` in C language. Integers are casted to real numbers automatically. So 1 and 1.0 are the same value. Numbers begin digits, not signs (`+` or `-`).

- 100, 20.34 and 0.01 are numbers
- +100 isn't a number. It causes syntax error. Use 100 instead.
- -100 isn't a number. But turtle recognizes it unary minus and a number 100. So turtle calculate it and the result is -100.
- 100 + -20: This is recognized 100 + (- 20). However, using bracket, 100 + (-20), is better for easy reading.

```
distance = 100
fd distance
```

A value 100 is assigned to the variable `distance` in the first line. Assignment is a statement. Most of statements begin with commands like `fd`. Assignment is the only exception.

The example above draws a line segment of 100 pixels long.

You can use variables in expressions. There are 8 kinds of calculations available.

- addition: x + y
- subtraction: x - y
- multiplication: x * y
- division: x / y
- unary minus: - x
- logical equal: x = y. This symbol `=` works as `==` in C language.
- greater than: x > y
- less than: x < y

The last three symbols are mainly used in the condition of if statement.

Variables are registered to a symbol table when it is assigned a value for the first time. Evaluating a variable before the registration isn't allowed and occurs an error.

A.7 If statement

Turtle language has very simple if statement.

```
if (x > 50) {
  fd x
}
```

There is no else part.

A.8 Loop

Turtle has very simple loop statement. It is `rp` (RePeat) statement.

```
rp (4) {
  fd 100
  tr 90
}
```

The program repeats the statements in the brace four times.

A.9 Procedures

Procedures are similar to functions in C language. The difference is that procedures don't have return values.

```
dp triangle (side) {
  fd side
  tr 120
  fd side
  tr 120
  fd side
}

triangle (100)
```

`dp` (Define Procedure) is a key word followed by procedure name, parameters, and body. Procedure names start alphabet followed by alphabet or digit. Parameters are a list of variables. For example

```
dp abc (a) { ... ... }
dp abc (a, b) { ... ... }
dp abc (a, b, c) { ... ... }
```

Body is a sequence of statements. The statements aren't executed when the procedure is defined. They will be executed when the procedure is called later.

Procedures are called by the name followed by arguments.

```
dp proc (a, b, c) { ... ... }

proc (100, 0, -20*3)
```

The number of parameters and arguments must be the same. Arguments can be any expressions. When you call a procedure, brackets following the procedure name must exist even if the procedure has no argument.

Procedure names and variable names don't conflict.

```
dp a () {fd a}
a=100
a ()
```

This is a correct program.

- 1: Defines a procedure `a`. A variable `a` is in its body.
- 2: Assigns 100 to a variable `a`.
- 3: Procedure `a` is called.

However, using the same name to a procedure and variable makes confusion. You should avoid that.

A.10 Recursive call

Procedures can be called recursively.

```
dp repeat (n) {
  n = n - 1
  if (n < 0) {
    rt
  }
  fd 100
  tr 90
  repeat (n)
}

repeat (4)
```

Repeat is called in the body of repeat. The call to itself is a recursive call. Parameters are created and set each time the procedure is called. So, parameter `n` is 4 at the first call but it is 3 at the second call. Every time the procedure is called, the parameter `n` decreases by one. Finally, it becomes less than zero, then the procedures return.

The program above draws a square.

It shows that we can program loop with a recursive call.

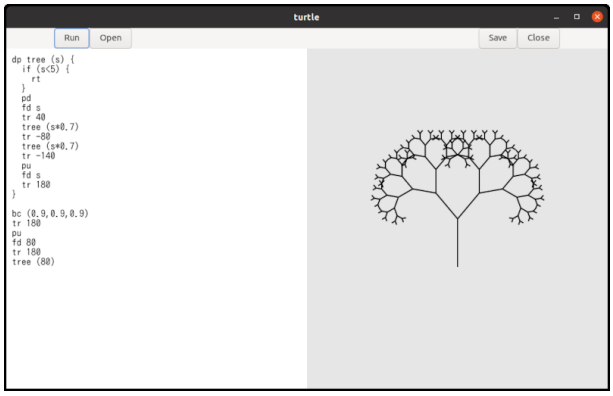


Figure 70: Tree

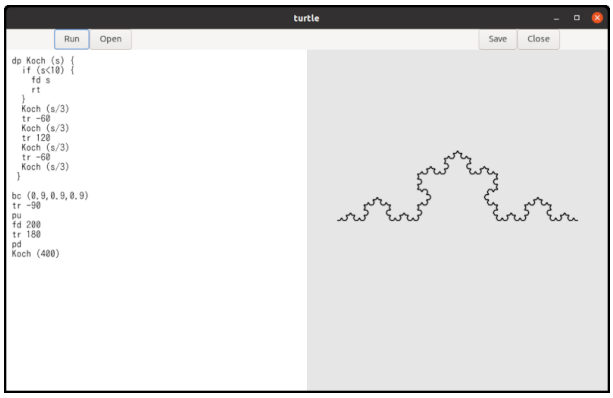


Figure 71: Koch curve

A.11 Fractal curves

Recursive call can be applied to draw fractal curves. Fractal curves appear when a procedure is applied to it repeatedly. The procedure replaces a part of the curve with the contracted curve.

This shape is called tree. The basic pattern of this shape is a line segment. It is the first stage. The second stage adds two shorter line segments at the endpoint of the original segment. The new segment has 70 percent length to the original segment and the orientation is +30 or -30 degrees different. The third stage adds two shorter line segments to the second stage line segments. And repeats it several times.

This repeating is programmed by recursive call. Two more examples are shown here. They are Koch curve and Square Koch curve.

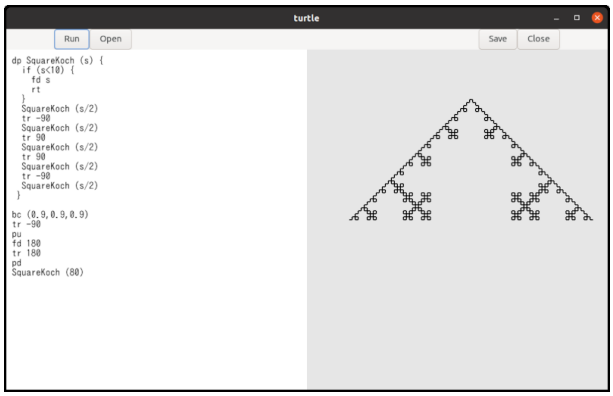


Figure 72: Square Koch curve

A.12 Tokens and punctuations

The following is the list of tokens.

Keywords:

- pu: pen up
- pd: pen down
- pw: pen width = line width
- fd: forward
- tr: turn right
- tl: turn left
- bc: background color
- fc: foreground color
- if: if statement
- rt: return
- rs: reset
- rp: repeat
- dp: define procedure

identifiers and numbers:

- identifier: This is used for the name of variables, parameters and procedures. It is expressed `[a-zA-Z][a-zA-Z0-9]*` in regular expression.
- number: This is expressed `(0|[1-9][0-9]*)(\.[0-9]+)?` in regular expression. It doesn't have `+` or `-` sign because they bring some syntactic confusion. However negative number such as `-10` can be recognized as unary minus and a number.

Symbols for expression

- `=`
- `>`
- `<`
- `+`
- `-`
- `*`
- `/`
- `(`
- `)`

Delimiters

- `(`
- `)`
- `{`
- `}`
- `,`

Comments and spaces:

- comment: This is characters between `#` and new line. If a comment is at the end of the file, the trailing new line can be left out.
- white space:
- horizontal tab: tab is recognized as eight spaces.
- new line: This is the end of a line.

These characters are used to separate tokens explicitly. They doesn't have any syntactic meaning and are ignored by the parser.

A.13 Grammar

```
program:
  statement
| program statement
;
```

```
statement:
  primary_procedure
| procedure_definition
;

primary_procedure:
  pu
| pd
| pw expression
| fd expression
| tr expression
| tl expression
| bc '(' expression ',' expression ',' expression ')'
| fc '(' expression ',' expression ',' expression ')'
| ID '=' expression    /* ID is an identifier which is a name of variable */
| if '(' expression ')' '{' primary_procedure_list '}'
| rt
| rs
| rp '(' expression ')' '{' primary_procedure_list '}'
| ID '(' ')'    /* ID is an identifier which is a name of procedure */
| ID '(' argument_list ')'    /* the same as above */
;

procedure_definition:
  dp ID '('  ')' '{' primary_procedure_list '}'
| dp ID '(' parameter_list ')' '{' primary_procedure_list '}'
;

parameter_list:
  ID
| parameter_list ',' ID
;

argument_list:
  expression
| argument_list ',' expression
;

primary_procedure_list:
  primary_procedure
| primary_procedure_list primary_procedure
;

expression:
  expression '=' expression
| expression '>' expression
| expression '<' expression
| expression '+' expression
| expression '-' expression
| expression '*' expression
| expression '/' expression
| '-' expression %prec UMINUS
| '(' expression ')'
| ID
| NUM    /* NUM is a number */
;
```

B TfeTextView API reference

B.1 Description

TfeTextView is a child object of GtkTextView. If its contents comes from a file, it holds the pointer to the GFile. Otherwise, the pointer is NULL.

B.2 Hierarchy

```
GObject -- GInitiallyUnowned -- GtkWidget -- GtkTextView -- TfeTextView
```

B.3 Ancestors

- GtkTextView
- GtkWidget
- GInitiallyUnowned
- GObject

B.4 Constructors

- tfe_text_view_new ()
- tfe_text_view_new_with_file ()

B.5 Instance methods

- tfe_text_view_get_file ()
- tfe_text_view_open ()
- tfe_text_view_save ()
- tfe_text_view_saveas ()

B.6 Signals

- Tfe.TextView::change-file
- Tfe.TextView::open-response

B.7 API for constructors, instance methods and signals

constructors

B.7.1 tfe_text_view_new()

```
GtkWidget *
tfe_text_view_new (void);
```

Creates a new TfeTextView instance and returns the pointer to it as GtkWidget. If an error happens, it returns `NULL`.

Return value

- a new TfeTextView.

B.7.2 tfe_text_view_new_with_file()

```
GtkWidget *
tfe_text_view_new_with_file (GFile *file);
```

Creates a new TfeTextView, reads the contents of the `file` and set it to the GtkTextBuffer corresponds to the newly created TfeTextView. Then returns the pointer to the TfeTextView as GtkWidget. If an error happens, it returns `NULL`.

Parameters

- file: a GFile

Return value

- a new TfeTextView.

Instance methods

B.7.3 tfe_text_view_get_file()

```
GFile *
tfe_text_view_get_file (TfeTextView *tv);
```

Returns the copy of the GFile in the TfeTextView.

Parameters

- tv: a TfeTextView

Return value

- the pointer to the GFile

B.7.4 tfe_text_view_open()

```
void
tfe_text_view_open (TfeTextView *tv, GtkWidget *win);
```

Shows a file chooser dialog so that a user can choose a file to read. Then, read the file and set the buffer with the contents. This function doesn't return the I/O status. Instead, the status is informed by `open-response` signal. The caller needs to set a handler to this signal in advance.

parameters

- tv: a TfeTextView
- win: the top level window

B.7.5 tfe_text_view_save()

```
void
tfe_text_view_save (TfeTextView *tv);
```

Saves the contents of the buffer to the file. If `tv` holds a GFile, it is used. Otherwise, this function shows a file chooser dialog so that the user can choose a file to save.

Parameters

- tv: a TfeTextView

B.7.6 tfe_text_view_saveas()

```
void
tfe_text_view_saveas (TfeTextView *tv);
```

Saves the contents of the buffer to a file. This function shows file chooser dialog so that a user can choose a file to save.

Parameters

- tv: a TfeTextView

Signals

B.7.7 change-file

```
void
user_function (TfeTextView *tv,
               gpointer user_data)
```

Emitted when the GFile in the TfeTextView object is changed. The signal is emitted when:

- a new file is opened and read
- a user chooses a file with the file chooser dialog and save the contents.

B.7.8 open-response

```
void
user_function (TfeTextView *tv,
               TfeTextViewOpenResponseType response-id,
               gpointer user_data)
```

Emitted after the user calls `tfe_text_view_open`. This signal informs the status of file I/O operation.

Enumerations

B.7.9 TfeTextViewOpenResponseType

Predefined values for the response id given by `open-response` signal.

Members:

- TFE_OPEN_RESPONSE_SUCCESS: The file is successfully opened.
- TFE_OPEN_RESPONSE_CANCEL: Reading file is canceled by the user.
- TFE_OPEN_RESPONSE_ERROR: An error happened during the opening or reading process.

C How to build Gtk4-Tutorial

C.1 Quick start guide

1. You need linux operationg system, ruby, rake, pandoc and latex system.
2. download this repository and uncompress the file.
3. change your current directory to the top directory of the source files.
4. type `rake html` to create html files. The files are created under the `docs` directory.
5. type `rake pdf` to create pdf a file. The file is created under the `latex` directory.

C.2 Prerequisites

- Linux operationg system. The programs in this repository has been tested on Ubuntu 21.04.
- Download the files in the repository. There are two ways to download.
  1. Use git. Type `git clone https://github.com/ToshioCP/Gtk4-tutorial.git` on the command-line.
  2. Download a zip file. Click on the `Code` button (green button) on the top page of the repository. Then, click "Download ZIP".
- Ruby and rake.
- Pandoc. It is used to convert markdown to html and/or latex.
- Latex system. Texlive2020 or later version is recommended. It is used to generate the pdf file.

C.3 GitHub Flavored Markdown

When you see gtk4_tutorial GitHub repository, you'll find the contents of the `Readme.md` file. This file is written in markdown language. Markdown files have `.md` suffix.

There are several kinds of markdown language. `Readme.md` uses 'GitHub Flavored Markdown', which is often shortened as GFM. Markdown files in the `gfm` directory are written in GFM. If you are not familiar with it, refer to the page GitHub Flavor Markdown spec.

C.4 Pandoc's markdown

This tutorial also uses another markdown – 'pandoc's markdown'. Pandoc is a converter between markdown, html, latex, word docx and so on. This type of markdown is used to convert markdown to html and/or latex.

C.5 .Src.md file

.Src.md file has ".src.md" suffix. The syntax of .src.md file is similar to markdown but it has a special command which isn't included in markdown syntax. It is @@@ command. The command starts with a line begins with "@@@" and ends with a line "@@@". For example,

```
@@@include
tfeapplication.c
@@@
```

There are four types of @@@ command.

C.5.1 @@@include

This type of @@@ command starts with a line "@@@include".

```
@@@include
tfeapplication.c
@@@
```

This command replaces itself with the texts read from the C source files surrounded by `@@@include` and `@@@`. If a function list follows the filename, only the functions are read.

```
@@@include
tfeapplication.c main startup
@@@
```

The command above is replaced by the contents of `main` and `startup` functions in the file `tfeapplication.c`.

Other language's source files are also possible. The following example shows that the ruby file 'lib_src2md.rb' is inserted by the command.

```
@@@include
lib_src2md.rb
@@@
```

You can't specify functions or methods unless the file is C source.

The inserted text is converted to fence code block. Fence code block begins with `~~~` and ends with `~~~`. The contents are displayed verbatim. `~~~` is look like a fence so the block is called "fence code block".

If the target markdown is GFM, then an info string can follow the beginning fence. The following example shows that the @@@ command includes a C source file `sample.c`.

```
$ cat src/sample.c
int
main (int argc, char **argv) {
  ... ...
}
$cat src/sample.src.md
  ... ...
@@@include -N
sample.c
@@@
  ... ...
$ ruby src2md.rb src/sample.src.md
$ cat gfm/sample.md
  ... ...
~~~C
int
main (int argc, char **argv) {
  ... ...
}
~~~
  ... ...
```

Info strings are usually languages like C, ruby, xml and so on. This string is decided with the filename extension.

- `.c` => C
- `.rb` => ruby
- `.xml` => xml

The supported languages are written in the `lang` method in `lib/lib_src2md.rb`.

A line number will be inserted at the top of each line in the code block. If you don't want to insert it, give "-N" option to @@@include command.

Options:

- `-n`: Inserts a line number at the top of each line (default).
- `-N`: No line number is inserted.

The following shows that the line numbers are inserted at the beginning of each line.

```
$cat src/sample.src.md
  ... ...
@@@include
sample.c
@@@
  ... ...
$ ruby src2md.rb src/sample.src.md
$ cat gfm/sample.md
  ... ...
~~~C
 1 int
 2 main (int argc, char **argv) {
  ... ...
14 }
~~~
  ... ...
```

If a markdown is an intermediate file to html, another type of info string follows the fence. If @@@include command doesn't have -N option, then the generated markdown is:

```
~~~{.C .numberLines}
int
main (int argc, char **argv) {
  ... ...
}
~~~
```

The info string `.C` specifies C language. The info string `.numberLines` is a class of the pandoc markdown. If the class is given, pandoc generates CSS to insert line numbers to the source code in the html file. That's why the fence code block in the markdown doesn't have line numbers, which is different from gfm markdown. If `-N` option is given, then the info string is `{.C}` only.

If a markdown is an intermediate file to latex, the same info string follows the beginning fence.

```
~~~{.C .numberLines}
int
main (int argc, char **argv) {
  ... ...
}
~~~
```

Rake uses pandoc with –listings option to convert the markdown to a latex file. The generated latex file uses 'listings package' to list source files instead of verbatim environment. The markdown above is converted to the following latex source file.

```
\begin{lstlisting}[language=C, numbers=left]
int
```

```
main (int argc, char **argv) {
  ... ...
}
\end{lstlisting}
```

Listing package can color or emphasize keywords, strings, comments and directives. But it doesn't really analyze the syntax of the language, so the emphasis tokens are limited.

@@@include command has two advantages.

1. Less typing.
2. You don't need to modify your .src.md file, even if the C source file is modified.

C.5.2 @@@shell

This type of @@@ command starts with a line begins with "@@@shell".

```
@@@shell
shell command
 ... ...
@@@
```

This command replaces itself with:

- the shell command
- the standard output from the shell command

For example,

```
@@@shell
wc Rakefile
@@@
```

This is converted to:

```
~~~
$ wc Rakefile
164  475 4971 Rakefile
~~~
```

C.5.3 @@@if series

This type of @@@ command starts with a line begins with "@@@if", and followed by "@@@elif", "@@@else" or "@@@end". This command is similar to "#if", "#elif", #else" and "#endif" directives in the C preprocessor. For example,

```
@@@if gfm
Refer to  [tfetextview API reference](tfetextview_doc.md)
@@@elif html
Refer to  [tfetextview API reference](tfetextview_doc.html)
@@@elif latex
Refer to tfetextview API reference in appendix.
@@@end
```

`@@@if` and `@@@elif` have conditions. They are `gfm`, `html` or `latex` so far.

- gfm: if the target is GFM
- html: if the target is html
- latex: if the target is pdf.

Other type of conditions may be available in the future version.

The code analyzing @@@if series command is rather complicated. It is based on the state diagram below.

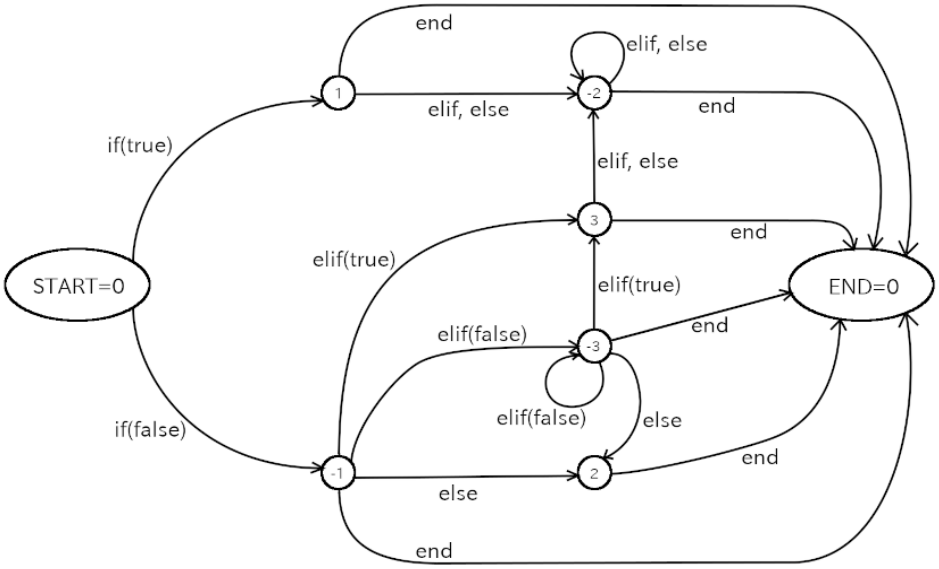


Figure 73: state diagram

C.5.4 @@@table

This type of @@@ command starts with a line begins with "@@@table". The contents of this command is a table of the GFM or pandoc's markdown. The command makes a table easy to read. For example, a text file `sample.md` has a table like this:

```
Price list

@@@table
|item|price|
|:---:|:---:|
|mouse|$10|
|PC|$500|
@@@
```

The command changes this into:

```
Price list

|item |price|
|:---:|:---:|
|mouse| $10 |
| PC  |$500 |
```

This command just changes the appearance of the table. There's no influence on html/latex files that is converted from the markdown. Notice that the command supports only the above type of markdown table format.

A script `mktbl.rb` supports this command. If you run the script like this:

```
$ ruby mktbl.rb sample.md
```

Then, the tables in 'sample.md' will be arranged. The script also makes a backup file `sample.md.bak`.

The task of the script seems easy, but the program is not so simple. The script `mktbl.rb` uses a library `lib/lib_src2md.rb`

@@@commands are effective in the whole text. This means you can't stop the @@@commands. But sometimes you want to show the commands literally like this document. One solution is to add four blanks

at the top of the line. Then @@@commands are not effective because @@@commands must be at the top of the line.

C.6 Conversion

The @@@ commands are carried out by a method `src2md`, which is in the file `lib/lib_src2md.rb`. This method converts `.src.md` file into `.md` file. In addition, some other conversions are made by `src2md` method.

- Relative links are changed according to the change of the base directory.
- Size option in an image link is removed when the destination is GFM or html.
- Relative link is removed except .src.md files when the destination is html.
- Relative link is removed when the destination is latex.

The order of the conversions are:

1. @@@if
2. @@@table
3. @@@include
4. @@@shell
5. others

There is the `src2md.rb` file in the top directory of this repository. It just invokes the method `src2md`. In the same way, the method is called in the action in the `Rakefile`.

C.7 Directory structure

There are seven directories under `gtk4_tutorial` directory. They are `gfm`, `docs`, `latex`, `src`, `image`, `test` and `lib`. Three directories `gfm`, `docs` and `latex` are the destination directories for GFM, html and latex files respectively. It is possible that these three directories don't exist before the conversion.

- src: This directory contains .src.md files and C-related source files.
- image: This directory contains image files like png or jpg.
- gfm: `rake` converts .src.md files to GFM files and store them in this directory.
- docs: `rake html` will convert .src.md files to html files and store them in this directory.
- latex: `rake pdf` will convert .src.md files to latex files and store them in this directory. Finally it creates a pdf file in `latex` directory.
- lib: This directory includes ruby library files.
- test: This directory contains test files. The tests are carried out by typing `rake test` on the terminal.

C.8 Src directory and the top directory

Src directory contains .src.md files and C-related source files. The top directory, which is gtk_tutorial directory, contains `Rakefile`, `src2md.rb` and some other files. When `Readme.md` is generated, it will be located at the top directory. `Readme.md` has title, abstract, table of contents with links to GFM files.

Rakefile describes how to convert .src.md files into GFM, html and/or pdf files. Rake carries out the conversion according to the `Rakefile`.

C.9 The name of files in src directory

Files in `src` directory are an abstract, sections of the document and other .src.md files. An `abstract.src.md` contains the abstract of this tutorial. Each section filename is "sec", number of the section and ".src.md" suffix. For example, "sec1.src.md", "sec5.src.md" or "sec12.src.md". They are the files correspond to the section 1, section 5 and section 12 respectively.

C.10 C source file directory

Most of .src.md files have `@@@include` commands and they include C source files. Such C source files are located in the subdirectories of `src` directory.

Those C files have been compiled and tested. When you compile source files, some auxiliary files and a target file like `a.out` are created. Or `_build` directory is made when `meson` and `ninja` is used when compiling. Those files are not tracked by `git` because they are specified in `.gitignore`.

The name of the subdirectories should be independent of section names. It is because of renumbering, which will be explained in the next subsection.

C.11 Renumbering

Sometimes you might want to insert a new section. For example, you want to insert it between section 4 and section 5. You can make a temporary section 4.5, that is a rational number between 4 and 5. However, section numbers are usually integer so section 4.5 must be changed to section 5. And the numbers of the following sections must be increased by one.

This renumbering is done by the `renumber` method in the `lib/lib_renumber.rb` file.

- It changes file names.
- If there are references (links) to sections in .src.md files, the section numbers will be automatically renumbered.

C.12 Rakefile

Rakefile is similar to Makefile but controlled by rake, which is a ruby script. Rakefile in this tutorial has the following tasks.

- md: generate GFM markdown files. This is the default.
- html: generate html files.
- pdf: generate latex files and a pdf file, which is compiled by lualatex.
- all: generate md, html and pdf files.
- clean: delete latex intermediate files.
- clobber: delete all the generated files.

Rake does renumbering before the tasks above.

C.13 Generate GFM markdown files

Markdown files (GFM) are generated by rake.

```
$ rake
```

This command generates `Readme.md` with `src/abstract.src.md` and titles of each `.src.md` file. At the same time, it converts each .src.md file into a GFM file under the `gfm` directory. Navigation lines are added at the top and bottom of each markdown section file.

You can describe width and height of images in .src.md files. For example,

```
![sample image](../image/sample_image.png){width=10cm height=6cm}
```

The size between left brace and right brace is used in latex file and it is not fit to GFM syntax. So the size will be removed in the conversion.

If a .src.md file has relative URL links, they will be changed by conversion. Because .src.md files are located under the `src` directory and GFM files are located under the `gfm` directory. That means the base directory of the relative link are different. For example, `[src/sample.c](sample.c)` is translated to `[src/sample.c](../src/sample.c)`.

If a link points another .src.md file, then the target filename will be changed to .md file. For example, `[Section 5](sec5.src.md)` is translated to `[Section 5](sec5.md)`.

If you want to clean the directory, that means remove all the generated markdown files, type `rake clobber`.

```
$ rake clobber
```

Sometimes this is necessary before generating GFM files.

```
$ rake clobber
$ rake
```

For example, if you append a new section and other files are still the same as before, `rake clobber` is necessary. Because the navigation of the previous section of the newly added section needs to be updated. If you don't do `rake clobber`, then it won't be updated because the the timestamp of .md file in gfm is newer than the one of .src.md file. In this case, using `touch` to the previous section .src.md also works to update the file.

If you see the GitHub repository (ToshioCP/Gtk4-tutorial), `Readme.md` is shown below the code. And `Readme.md` includes links to each markdown files. The repository not only stores source files but also shows the whole tutorial.

C.14 Generate html files

Src.md files can be translated to html files. You need pandoc to do this. Most linux distribution has pandoc package. Refer to your distribution document to install.

Type `rake html` to generate html files.

```
$ rake html
```

First, it generates pandoc's markdown files under `docs` directory. Then, pandoc converts them to html files. The width and height of image files are removed. Links to .src.md files will be converted like this.

```
[Section 5](sec5.src.md) => [Section 5](sec5.html)
```

Image files are copied to `docs/image` direcotiry and links to them will be converted like this:

```
[sample.png](../image/sample.png) => [sample.png](image/sample.png)
```

Other relative links will be removed.

`index.html` is the top html file. If you want to clean html files, type `rake clobber` or `cleanhtml`.

```
$ rake clobber
```

Every html file has a header (`<head> -- </head>`). It is created by pandoc with '-s' option. You can customize the output with your own template file for pandoc. Rake uses `lib/lib_mk_html_template.rb` to create its own template. The template inserts bootstrap CSS and Javascript through `jsDelivr`.

The `docs` directory contains all the necessary html files. They are used in the GitHub pages of this repository.

So if you want to publish this tutorial on your own web site, just upload the files in the `docs` directory to your site.

C.15 Generate a pdf file

You need pandoc to convert markdown files into latex source files.

Type `rake pdf` to generate latex files and finally make a pdf file.

```
$ rake pdf
```

First, it generates pandoc's markdown files under `latex` directory. Then, pandoc converts them into latex files. Links to files or directories are removed because latex doesn't support them. However, links to full URL and image files are kept. Image size is set with the size between the left brace and right brace.

```
![sample image](../image/sample_image.png){width=10cm height=6cm}
```

You need to specify appropriate width and height. It is almost `0.015 x pixels` cm. For example, if the width of an image is 400 pixels, the width in a latex file will be almost 6cm.

A file `main.tex` is the root file of all the generated latex files. It has `\input` commands, which inserts each section file, between `\begin{document}` and `\end{document}`. It also has `\input`, which inserts `helper.tex`, in the preamble. Two files `main.tex` and `helper.tex` are created by `lib/lib_gen_main_tex.rb`. It has a sample markdown code and converts it witn `pandoc -s`. Then, it extracts the preamble in the generated file and puts it into `helper.tex`. You can customize `helper.tex` by modifying `lib/lib_gen_main_tex.rb`.

Finally, lualatex compiles the `main.tex` into a pdf file.

If you want to clean `latex` directory, type `rake clobber` or `rake cleanlatex`

```
$ rake clobber
```

This removes all the latex source files and a pdf file.