# Rapport GTK

## Sources

**https://ocaml.org/learn/tutorials/introduction_to_gtk.html**
**https://docs.gtk.org/**
**https://docs.gtk.org/gobject/index.html**
**https://docs.gtk.org/glib/**
**https://fr.wikipedia.org/wiki/GTK_(bo%C3%AEte_%C3%A0_outils)**
**https://gtk.developpez.com/cours/gtk2/**

GTK est multi plateformes grâce à des liaisons entre langages(Language Binding)

Parmi les langages on trouve C++, Go, Ocaml, Haskell, Javascript, Kotlin, Python…

## GTK : Les bases

Les objets graphiques sont appelés widgets. Des structures définissent les propriétés de l'objet. De nombreuses fonctions sont associées à ces widgets.

Il y' a une notion d'héritage et une hiérarchie bien définie, qu'on peut constater notamment lorsqu'on utilise les macros de conversion de GTK.

## GTK et les objets

Lorsqu'on souhaite développer des applications GTK, on se rend assez vite compte que cette ensemble de bibliothèques logicielles a été conçue avec comme philosophie la programmation orientée objet, alors même que le langage C ne permet pas ce genre d'approche. De ce constat on peut alors se demander : Par quels moyens GTK s'approprie la POO ?

GTK tient son système de type de Glib. Cette bibliothèque C fournit un framework orienté-objet pour le C. On peut notamment trouver dans sa contribution:
-Un système de type générique pour enregistrer arbitrairement des « objets ». Ce système prend en compte la création, l'initialisation et la gestion de la mémoire de ces objets, ainsi que des relations d'héritages qui peuvent être définies.
-Un exemple d'implémentation de type fondamental pour sur lequel baser les hiérarchies d'objets : GObject

GObject est le point central permettant la simulation de POO en C avec Glib. Nous allons maintenant plonger plus en profondeur dans les entrailles de Glib pour essayer de comprendre comment tout ça fonctionne.

**Généralités.**

GObject et son système de type bas-niveau GType fournissent des API orientés objets en C ainsi qu'une API facilitant le binding pour d'autres langages, compilés ou interprétés.

**Le système de typage dynamique de Glib.**

Un type tel que manipulé par Glib est bien plus générique que ce à quoi on peut penser. Ci-dessous la structure et les fonctions utilisées pour enregistrer de nouveaux types dans le système de type.

```
typedef struct _GTypeInfo              GTypeInfo;
struct _GTypeInfo
{
  /* interface types, classed types, instantiated types */
  guint16               class_size;

  GBaseInitFunc         base_init;
  GBaseFinalizeFunc     base_finalize;

  /* classed types, instantiated types */
  GClassInitFunc        class_init;
  GClassFinalizeFunc    class_finalize;
  gconstpointer         class_data;

  /* instantiated types */
  guint16               instance_size;
  guint16               n_preallocs;
  GInstanceInitFunc     instance_init;

  /* value handling */
  const GTypeValueTable *value_table;
};

GType
g_type_register_static (GType            parent_type,
                        const gchar     *type_name,
                        const GTypeInfo *info,
                        GTypeFlags       flags);

GType
g_type_register_fundamental (GType                       type_id,
                             const gchar                *type_name,
                             const GTypeInfo            *info,
                             const GTypeFundamentalInfo *finfo,
                             GTypeFlags                  flags);
```

GObject est la classe de base des objets instanciables. Elle implémente :
-Une gestion de la mémoire avec comptage de référence.
-Un mécanisme de construction/destruction des instances.

-Des propriétés génériques et les setters et getters associés.
-Des signaux.

**<u>Les objets</u>**

Les types enregistrés avec une classe et déclarés instanciables sont ceux qui se rapprochent le plus des objets dans le sens de la POO.Voici un exemple de ce qu'il est possible de faire.

```c
typedef struct {
  GObject parent_instance;

  /* instance members */
  char *filename;
} ViewerFile;

typedef struct {
  GObjectClass parent_class;

  /* class members */

  /* the first is public, pure and virtual */
  void (*open)  (ViewerFile  *self,
                 GError     **error);

  /* the second is public and virtual */
  void (*close) (ViewerFile  *self,
                 GError     **error);
} ViewerFileClass;

#define VIEWER_TYPE_FILE (viewer_file_get_type ())

GType
viewer_file_get_type (void)
{
  static GType type = 0;
  if (type == 0) {
    const GTypeInfo info = {
      .class_size = sizeof (ViewerFileClass),
      .base_init = NULL,
      .base_finalize = NULL,
      .class_init = (GClassInitFunc) viewer_file_class_init,
      .class_finalize = NULL,
      .class_data = NULL,
      .instance_size = sizeof (ViewerFile),
      .n_preallocs = 0,
      .instance_init = (GInstanceInitFunc) viewer_file_init,
    };
    type = g_type_register_static (G_TYPE_OBJECT,
                                   "ViewerFile",
                                   &info, 0);
  }
  return type;
}
```

Lors du premier appel à **viewer_file_get_type**, le type **ViewerFile** sera enregistré dans le système de type comme héritant du type **G_TYPE_OBJECT**.

Chaque objet doit définir deux structures : sa structure de classe et sa structure d'instance. Toutes les structures de classe doivent contenir comme premier membre une structure **GTypeClass**. Toutes les structures d'instance doivent contenir comme premier membre une structure **GtypeInstance.**

Ces contraintes permettent au système de type de s'assurer que chaque instance d'objet (identifiée par un pointeur vers la structure d'instance de l'objet) contient dans ses premiers octets un pointeur vers la structure de classe de l'objet.

**Instanciation des objets.**

La famille de fonctions g_object_new() peut être utilisée pour instancier un Gtype qui descend d’un type de base Gobject. Les objets descendants de Gobject sont autorisés à redéfinir le constructeur.

Exemple

```c
#define VIEWER_TYPE_FILE viewer_file_get_type ()
G_DECLARE_FINAL_TYPE (ViewerFile, viewer_file, VIEWER, FILE, GObject)

struct _ViewerFile
{
  GObject parent_instance;

  /* instance members */
  char *filename;
  guint zoom_level;
};

/* will create viewer_file_get_type and set viewer_file_parent_class */
G_DEFINE_TYPE (ViewerFile, viewer_file, G_TYPE_OBJECT)

static void
viewer_file_constructed (GObject *obj)
{
  /* update the object state depending on constructor properties */

  /* Always chain up to the parent constructed function to complete object
   * initialisation. */
  G_OBJECT_CLASS (viewer_file_parent_class)->constructed (obj);
}

static void
viewer_file_finalize (GObject *obj)
{
  ViewerFile *self = VIEWER_FILE (obj);

  g_free (self->filename);

  /* Always chain up to the parent finalize function to complete object
   * destruction. */
  G_OBJECT_CLASS (viewer_file_parent_class)->finalize (obj);
}

static void
viewer_file_class_init (ViewerFileClass *klass)
{
  GObjectClass *object_class = G_OBJECT_CLASS (klass);

  object_class->constructed = viewer_file_constructed;
  object_class->finalize = viewer_file_finalize;
}

static void
viewer_file_init (ViewerFile *self)
{
  /* initialize the object */
}
```

```c
ViewerFile *file = g_object_new (VIEWER_TYPE_FILE, NULL);
```

Une fois que **g_object_new()** a obtenu une référence à une structure de classe initialisée, il invoque son constructeur pour créer une instance du nouvel objet, si le constructeur a été remplacé dans **viewer_file_class_init**. Les constructeurs surchargés doivent s'enchaîner avec le constructeur de leur parent. Pour trouver la classe parent et enchaîner avec le constructeur de la classe parent, il est possible d'utiliser le pointeur **viewer_file_parent_class** qui a été configuré par la macro **G_DEFINE_TYPE**.

A un moment donné, **g_object_constructor** est invoqué par le dernier constructeur de la chaîne. La fonction va allouer de la mémoire pour l'objet par l'intermédiaire de **g_type_create_instance()** ce qui signifie que la fonction **instance_init** est invoquée à ce stade si elle a été enregistrée. Après le retour de **instance_init**, l'objet est entièrement initialisé et devrait être prêt à recevoir ses méthodes appelées par l'utilisateur. Lorsque **g_type_create_instance()** termine, **g_object_constructor** définit les propriétés de construction (c'est-à-dire les propriétés qui ont été données à **g_object_new()**) et retourne au constructeur de l'utilisateur.

Voici un tableau qui récapitule les fonctions invoquées par g_object_new() et l'ordre d'invocation associé.

| Invocation time | Function invoked | Function's parameters | Remark |
|---|---|---|---|
| First call to `g_object_new()` for target type | target type's `base_init` function | On the inheritance tree of classes from fundamental type to target type. `base_init` is invoked once for each class structure. | Never used in practice. Unlikely you will need it. |
| | target type's `class_init` function | On target type's class structure | Here, you should make sure to initialize or override class methods (that is, assign to each class' method its function pointer) and create the signals and the properties associated to your object. |
| | interface's `base_init` function | On interface's vtable | |
| | interface's `interface_init` function | On interface's vtable | |
| Each call to `g_object_new()` for target type | target type's class `constructor` method: `GObjectClass->constructor` | On object's instance | If you need to handle construct properties in a custom way, or implement a singleton class, override the constructor method and make sure to chain up to the object's parent class before doing your own initialization. In doubt, do not override the constructor method. |
| | type's `instance_init` function | On the inheritance tree of classes from fundamental type to target type. The `instance_init` provided for each type is invoked once for each instance structure. | Provide an `instance_init` function to initialize your object before its construction properties are set. This is the preferred way to initialize a GObject instance. This function is equivalent to C++ constructors. |
| | target type's class `constructed` method: `GObjectClass->constructed` | On object's instance | If you need to perform object initialization steps after all construct properties have been set. This is the final step in the object initialization process, and is only called if the `constructor` method returned a new object instance (rather than, for example, an existing singleton). |

**Les propriétés des objets.**

Une fonctionnalité intéressante de **GObject** est son mécanisme de set/get générique pour les propriétés d'un objet. Quand l'objet est instancié on utilise l'handler **class_init** pour enregistrer les propriétés de l'objet avec **g_object_class_install_properties()**.

```c
// Implementation

typedef enum
{
  PROP_FILENAME = 1,
  PROP_ZOOM_LEVEL,
  N_PROPERTIES
} ViewerFileProperty;

static GParamSpec *obj_properties[N_PROPERTIES] = { NULL, };

static void
viewer_file_set_property (GObject      *object,
                          guint         property_id,
                          const GValue *value,
                          GParamSpec   *pspec)
{
  ViewerFile *self = VIEWER_FILE (object);

  switch ((ViewerFileProperty) property_id)
    {
    case PROP_FILENAME:
      g_free (self->filename);
      self->filename = g_value_dup_string (value);
      g_print ("filename: %s\n", self->filename);
      break;

    case PROP_ZOOM_LEVEL:
      self->zoom_level = g_value_get_uint (value);
      g_print ("zoom level: %u\n", self->zoom_level);
      break;

    default:
      /* We don't have any other property... */
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
      break;
    }
}

static void
viewer_file_get_property (GObject    *object,
                          guint       property_id,
                          GValue     *value,
                          GParamSpec *pspec)
{
  ViewerFile *self = VIEWER_FILE (object);

  switch ((ViewerFileProperty) property_id)
    {
    case PROP_FILENAME:
      g_value_set_string (value, self->filename);
      break;

    case PROP_ZOOM_LEVEL:
      g_value_set_uint (value, self->zoom_level);
      break;

    default:
      /* We don't have any other property... */
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, property_id, pspec);
      break;
    }
}

static void
viewer_file_class_init (ViewerFileClass *klass)
{
  GObjectClass *object_class = G_OBJECT_CLASS (klass);

  object_class->set_property = viewer_file_set_property;
  object_class->get_property = viewer_file_get_property;

  obj_properties[PROP_FILENAME] =
    g_param_spec_string ("filename",
                         "Filename",
                         "Name of the file to load and display from.",
                         NULL  /* default value */,
                         G_PARAM_CONSTRUCT_ONLY | G_PARAM_READWRITE);

  obj_properties[PROP_ZOOM_LEVEL] =
    g_param_spec_uint ("zoom-level",
                       "Zoom level",
                       "Zoom level to view the file at.",
                       0  /* minimum value */,
                       10 /* maximum value */,
                       2  /* default value */,
                       G_PARAM_READWRITE);

  g_object_class_install_properties (object_class,
                                     N_PROPERTIES,
                                     obj_properties);
}
```

**g_object_set_property()** s'assure d'abord qu'une propriété portant ce nom a été enregistrée dans l'handler **class_init** du fichier. Si c'est le cas, il parcourt la hiérarchie des classes, du type le plus dérivé le plus bas au type le plus haut dans la hiérarchie pour trouver la classe qui a enregistré cette propriété. Il essaie ensuite de convertir la **GValue** fournie par l'utilisateur en une **GValue** dont le type est celui de la propriété associée.

Si l'utilisateur fournit un char signé **GValue**, comme indiqué ici, et si la propriété de l'objet a été enregistrée en tant qu'int non signé, **g_value_transform()** essaiera de transformer le char signé d'entrée en un int non signé. La réussite de la transformation dépend de la disponibilité de la fonction de transformation requise.

Après transformation, la GValue est validée par **g_param_value_validate()** qui s'assure que les données de l'utilisateur stockées dans la GValue correspondent aux caractéristiques spécifiées par la propriété **GParamSpec**. Ici, le

**GParamSpec** qui a été fourni dans **class_init** a une fonction de validation qui s'assure que le **GValue** contient une valeur qui respecte les limites minimales et maximales du **GParamSpec**. Dans l'exemple, la **GValue** ne respecte pas ces contraintes (elle est fixée à 11, alors que le maximum est de 10). Ainsi, la fonction **g_object_set_property()** renverra une erreur.

Si la **GValue** de l'utilisateur avait été définie sur une valeur valide, **g_object_set_property()** aurait procédé à l'appel de la méthode **set_property** de l'objet. Ici, puisque l'implémentation de **ViewerFile** a remplacé cette méthode, l'exécution sauterait à **viewer_file_set_property** après avoir récupéré du **GParamSpec** le **param_id** qui avait été stocké par **g_object_class_install_property()**.

C'est assez rébarbatif de configurer des **GValue** à chaque fois que l'on veut modifier une propriété. C'est quelque chose qui est en réalité utilisé assez rarement, les fonctions **g_object_set_property()** et **g_object_get_property()** sont destinées à être utilisées pour faire des binding de langage. Pour le développement d'applications, il existe un moyen plus simple :

```
ViewerFile *file;
file = /* */;
g_object_set (G_OBJECT (file),
              "zoom-level", 6,
              "filename", "~/some-file.txt",
              NULL);
```

**Gestion des évenements avec GTK.**

Pour faire réagir une application aux actions de l'utilisateur on utilise des signaux.

Il y' a deux notions importantes :

-La boucle événementielle qui va intercepter le signal. (**gtk_main**)

-L'association d'une fonction au signal d'un widget

## LablGTK

Avec lablgtk ces objets sont définis comme suit : type 'a obj

Le type obj est polymorphique, mais il ne définit aucune manière de le créer

C'est parce que les instances du type obj sont des "sous classes" de GTKObject et sont donc créés par du code C

```
CAMLprim value
ml_gtk_toggle_button_new (value unit)
{
  return Val_GtkObject_sink ((GtkObject *) gtk_toggle_button_new ());
}
```

Le mot clef “CAMLprim” indique que la fonction sera “appelée” par du code caml, la fonction “gtk_toggle_button_new” est une fonction de GTK+.

```
module ToggleButton = struct
  include ToggleButton
  let create_check pl : toggle_button obj = Object.make "GtkCheckButton" pl
  external toggled : [>`toggle] obj -> unit
      = "ml_gtk_toggle_button_toggled"
end
```

Ici le mot clef “external” indique que la valeur sera le retour d’une fonction C

“ml_gtk_toggle_button_toggled” est le nom d’une fonction C

Ocaml ayant un système de type fort il est normalement impossible d’effectuer du “downcast”

Cependant ce type d'opération est nécessaire lors de l'utilisation de GTK.

Lablgtk apporte donc une solution à ce problème.

## Comparaison OCAML / C

Un programme en OCAML

```
open GMain
open GdkKeysyms

let locale = GtkMain.Main.init ()

let main () =
  let window = GWindow.window ~width:320 ~height:240
                              ~title:"Simple lablgtk program" () in
  let vbox = GPack.vbox ~packing:window#add () in
  window#connect#destroy ~callback:Main.quit;

  (* Menu bar *)
  let menubar = GMenu.menu_bar ~packing:vbox#pack () in
  let factory = new GMenu.factory menubar in
  let accel_group = factory#accel_group in
  let file_menu = factory#add_submenu "File" in

  (* File menu *)
  let factory = new GMenu.factory file_menu ~accel_group in
  factory#add_item "Quit" ~key:_Q ~callback: Main.quit;

  (* Button *)
  let button = GButton.button ~label:"Push me!"
                              ~packing:vbox#add () in
  button#connect#clicked ~callback: (fun () -> prerr_endline "Ouch!");

  (* Display the windows and enter Gtk+ main loop *)
  window#add_accel_group accel_group;
  window#show ();
  Main.main ()

let () = main ()
```

Le même en C

Éditeur de texte

Ouvrir

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

void on_activate_entry(GtkWidget *pEntry, gpointer data);
void on_click_button();

/* Définition des éléments du menu */
static GtkItemFactoryEntry menu_items[] = {
  { "/_File",         NULL,         NULL, 0, "<Branch>" },
  { "/File/Quit",     "<control>Q", gtk_main_quit, 0, NULL }
};

void get_main_menu( GtkWidget  *window,
                    GtkWidget **menubar )
{
  /* MENU */
  GtkItemFactory *item_factory;
  GtkAccelGroup *accel_group;
  gint nmenu_items = sizeof (menu_items) / sizeof (menu_items[0]);

  accel_group = gtk_accel_group_new ();

  item_factory = gtk_item_factory_new (GTK_TYPE_MENU_BAR, "<main>",
                                       accel_group);


  gtk_item_factory_create_items (item_factory, nmenu_items, menu_items, NULL);


  gtk_window_add_accel_group (GTK_WINDOW (window), accel_group);

  if (menubar)

    *menubar = gtk_item_factory_get_widget (item_factory, "<main>");
}


int main(int argc, char **argv)
{
    GtkWidget *pWindow;
    GtkWidget *pVBox;
    GtkWidget *pButton;
    GtkWidget *menubar;

    gtk_init(&argc, &argv);

    pWindow = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(pWindow), "Présentation");
    gtk_window_set_default_size(GTK_WINDOW(pWindow), 320, 240);
    g_signal_connect(G_OBJECT(pWindow), "destroy", G_CALLBACK(gtk_main_quit), NULL);

    pVBox = gtk_vbox_new(FALSE, 1);
    gtk_container_add(GTK_CONTAINER(pWindow), pVBox);
    gtk_widget_show(pVBox);

  get_main_menu (pWindow, &menubar);
  gtk_box_pack_start (GTK_BOX (pVBox), menubar, FALSE, TRUE, 0);
  gtk_widget_show (menubar);

    pButton = gtk_button_new_with_label("Copier");
    gtk_box_pack_start(GTK_BOX(pVBox), pButton, TRUE, FALSE, 0);


    /* Connexion du signal "clicked" du GtkButton */
    g_signal_connect(G_OBJECT(pButton), "clicked", G_CALLBACK(on_click_button), NULL);
    gtk_widget_show_all(pWindow);

    gtk_main();

    return EXIT_SUCCESS;
}

/* Fonction callback executee lors du signal "clicked" */
void on_click_button()
{
  fprintf(stderr, "ouch");
}
```