Premiers pas avec GTKAda

GTKAda est la boite à outil graphique en Ada basée sur GTK pour construire des applications portables sur la plupart des plateformes.

Sites :

- www.gtk.org
- www.adacore.com/gtkada

Sommaire

1. Introduction

Pour illustrer les exemples, j'utiliserai cette configuration pour illustrer les exemples : macOS 11, GNAT CE 2021, XNAdaLib-CE-2021.

Voir leur installation sur Blady :

- macOS : blady.pagesperso-orange.fr/liens.html#macosx
- GNAT : blady.pagesperso-orange.fr/creations.html#gnatosxinstall
- XNAdaLib : blady.pagesperso-orange.fr/creations.html#xadalib

GTKAda étant multi-plateforme ce n'est pas obligatoire mais je recommande d'utiliser des versions récentes.

Ne pas oublier de modifier les variables PATH et GPR_PROJECT_PATH pour avoir accès au compilateur GNAT et à la bibliothèque GTKAda. Positionner également la variable *xnadalib* avec le chemin de votre installation de GTKAda, avec ma configuration, je prends :

```
$ xnadalib=/usr/local/xnadalib-ce-2021
$ GPR_PROJECT_PATH=$xnadalib/lib/gnat
$ PATH=/opt/gnat-ce-2021:$PATH
```

Une documentation au format HTML est disponible dans les dossiers *share/doc/gtkada* :

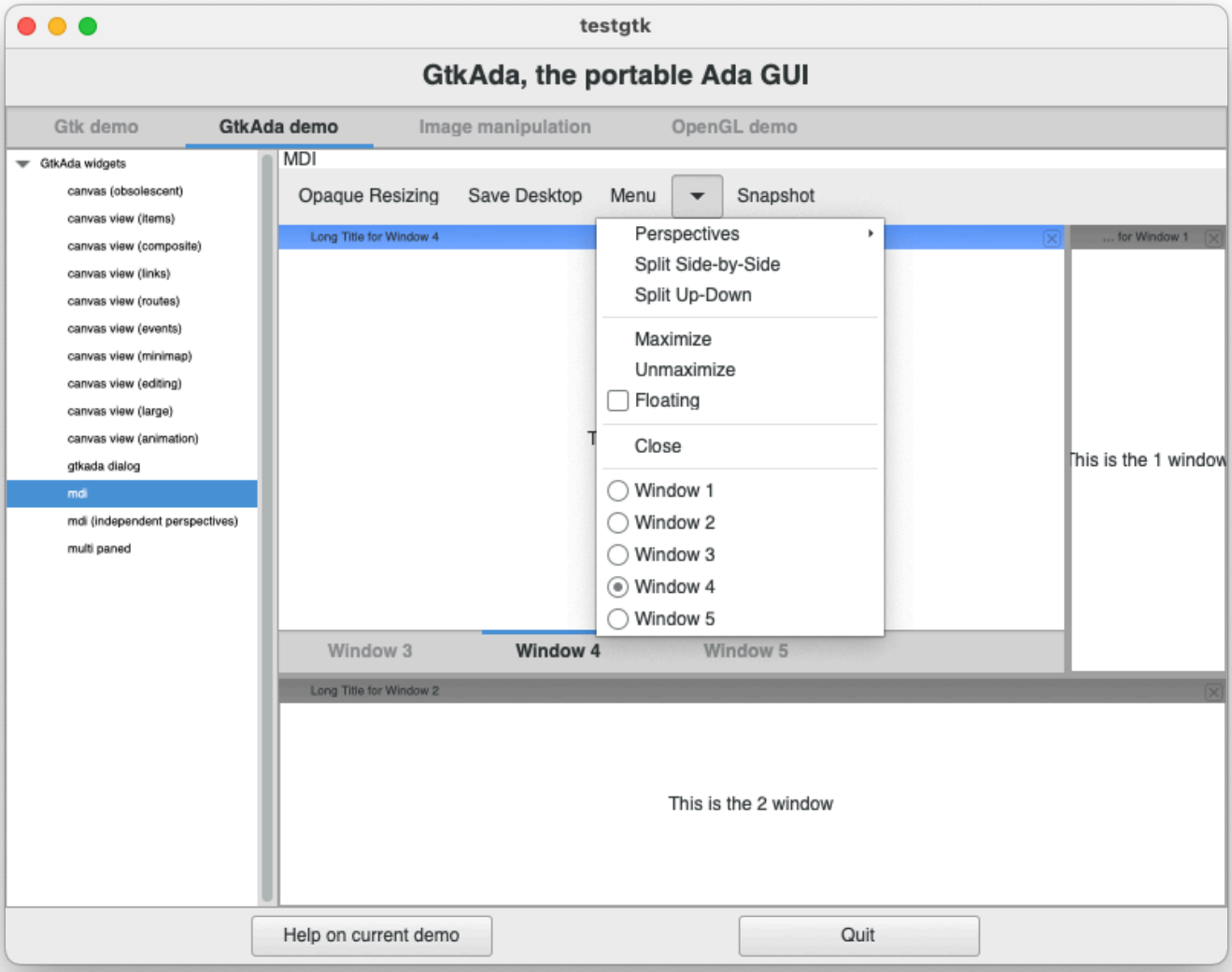
```
# Manuel utilisateur GTKAda :
$ open $xnadalib/share/doc/gtkada/gtkada_ug/index.html
# Manuel de référence GTKAda :
$ open $xnadalib/share/doc/gtkada/gtkada_rm/index.html
```

Le manuel de référence GTKAda est très complet avec un classement des objets graphiques par catégories, un aperçu graphique de chacun, un index. Une page est consacrée à chaque objet avec une description, les types, sous-programmes, signaux et propriétés déclarées ainsi qu'un exemple de programme dédié. Celui de GTK+ également mais dédié au langage C.

Des démonstrations sont présentes dans le dossier *share/examples/gtkada/testgtk* :

```
$ cd $xnadalib/share/examples/gtkada/testgtk
$ export XDG_DATA_DIRS=$xnadalib/share
$ ./testgtk
```

Le résultat :


testgtk
GtkAda, the portable Ada GUI
Gtk demo
GtkAda demo
Image manipulation
OpenGL demo
GtkAda widgets
canvas (obsolescent)
canvas view (items)
canvas view (composite)
canvas view (links)
canvas view (routes)
canvas view (events)
canvas view (minimap)
canvas view (editing)
canvas view (large)
canvas view (animation)
gtkada dialog
mdi
mdi (independent perspectives)
multi paned
MDI
Opaque Resizing
Save Desktop
Menu
Snapshot
Perspectives
Split Side-by-Side
Split Up-Down
Maximize
Unmaximize
Floating
Close
Window 1
Window 2
Window 3
Window 4
Window 5
Long Title for Window 4
Window 3
Window 4
Window 5
... for Window 1
This is the 1 window
Long Title for Window 2
This is the 2 window
Help on current demo
Quit

2. Utilisation avec le Terminal

Voici un programme très simple qui n'affiche qu'une fenêtre avec un texte :
(un clic sur le bouton de fermeture de la fenêtre termine le programme)

Le code source hello_callback.ads (spécification de l'action pour quitter) :

```
with Gtk.Window, Gtk.Handlers;
package Hello_Callback is
   package Window_Cb is new Gtk.Handlers.Return_Callback (
      Gtk.Window.Gtk_Window_Record,
      Boolean);
   function Quit
     (Widget : access Gtk.Window.Gtk_Window_Record'Class)
      return   Boolean;
end Hello_Callback;
```

Le code source hello_callback.adb (code de l'action pour quitter) :

```
with Gtk.Main;
package body Hello_Callback is
   function Quit
     (Widget : access Gtk.Window.Gtk_Window_Record'Class)
      return   Boolean
   is
      pragma Unreferenced (Widget);
   begin
      Gtk.Main.Main_Quit;
      return False;
   end Quit;
end Hello_Callback;
```

Le code source hello.adb (code du programme principal) :

```
with Gtk.Main, Gtk.Window, Gtk.Enums, Gtk.Label;
with Hello_Callback;
procedure Hello is
   Main_Window : Gtk.Window.Gtk_Window;
   Label       : Gtk.Label.Gtk_Label;
begin
   Gtk.Main.Init;
   Gtk.Window.Gtk_New
     (Window   => Main_Window,
      The_Type => Gtk.Enums.Window_Toplevel);
   Gtk.Window.Set_Title (Window => Main_Window, Title => "Hello");
   Gtk.Window.Set_Default_Size
     (Window => Main_Window,
      Width  => 230,
      Height => 150);
   Gtk.Window.Set_Position
     (Window   => Main_Window,
      Position => Gtk.Enums.Win_Pos_Center);
   Gtk.Label.Gtk_New (Label, "Hello with GTKAda.");
   Gtk.Window.Add (Main_Window, Label);
```

```
   Hello_Callback.Window_Cb.Connect
     (Main_Window,
      "delete_event",
      Hello_Callback.Quit'Access);
   Gtk.Window.Show_All (Main_Window);
   Gtk.Main.Main;
end Hello;
```

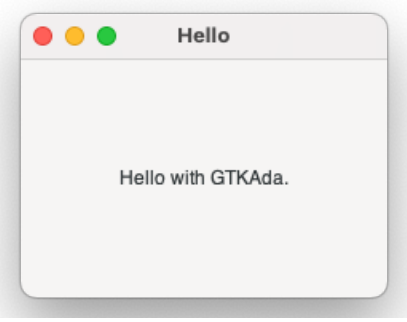
Le code source hello.gpr (le projet GPR) :

```
with "gtkada.gpr";
project Hello is
   for Main use ("hello.adb");
end Hello;
```

Saisir les commandes suivantes dans le Terminal :

```
$ gprbuild -P hello.gpr
$ ./hello
```

Et voilà le résultat :



La compilation s'est effectuée avec comme d'habitude avec *gprbuild* avec le projet *hello.gpr* qui référence la bibliothèque GTKAda.

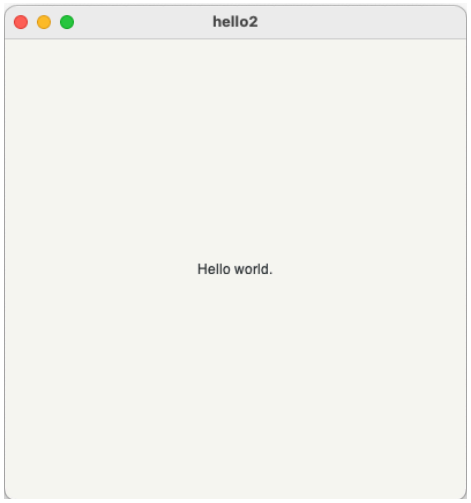
Le code source complet de Hello est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Hello

3. Utilisation avec GPS / GnatStudio

Au lancement de GPS choisir *Create new project from template* ou sélectionner le menu *Project->New from template* puis *GtkAda->Simple window*. Donner le nom *Hello2* au projet ainsi qu'au nom du programme principal, sélectionner le dossier hôte du projet (qui doit être déjà créé) puis cliquer sur *Apply*. Un programme tout prêt utilisant GTKAda s'affiche dans le dossier *src*.

Pour ajouter manuellement la bibliothèque GTKAda à un projet déjà existant, sélectionner le projet dans la Project View faire un clic droit et dans le menu contextuel sélectionner *Project >Edit source file* et ajouter *with "gtkada";* tout au début du fichier projet, sauvegarder puis sélectionner le menu *Project->Reload Project*.

Faites comme habituellement *Build* et *Run*, et voilà :



Le code source complet de Hello2 est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Hello2

Avertissement : Pour son fonctionnement GPS positionne les variables d'environnement spécifiques pour GTK vers son environnement d'installation. En exécutant votre programme depuis GPS, cela peut en perturber le fonctionnement. Le contournement est de repositionner les valeurs d'origines sauvegardées par le script de lancement de GPS avant d'exécuter *Gtk.Main.Init;* :

Le code source restore_gps_startup_values.adb (restore les valeurs masquées par GPS) :

```
with Ada.Environment_Variables;
with Ada.Strings.Fixed;
procedure Restore_GPS_Startup_Values is
  procedure Internal_RGSV (Saved_Var_Name, Saved_Var_Value : String) is
    GPS_STARTUP : constant String  := "GPS_STARTUP_";
    Ind         : constant Natural :=
      Ada.Strings.Fixed.Index (Saved_Var_Name, GPS_STARTUP);
    Oringinal_Var_Name : constant String :=
      Saved_Var_Name (Ind + GPS_STARTUP'Length .. Saved_Var_Name'Last);
  begin
    if Ind > 0
      and then Ada.Environment_Variables.Exists (Oringinal_Var_Name)
    then
      if Saved_Var_Value /= "" then
```

```
          Ada.Environment_Variables.Set
            (Oringinal_Var_Name,
             Saved_Var_Value);
        else
          Ada.Environment_Variables.Clear (Oringinal_Var_Name);
        end if;
      end if;
  end Internal_RGSV;
begin
  Ada.Environment_Variables.Iterate (Internal_RGSV'Access);
end Restore_GPS_Startup_Values;
```

Le code source hello3.adb (programme principal avec l'appel à Restore_GPS_Startup_Values) :

```
<...>
with Restore_GPS_Startup_Values;
procedure Hello3 is
   Main_Window : Gtk.Window.Gtk_Window;
   Label       : Gtk.Label.Gtk_Label;
begin
  Restore_GPS_Startup_Values;
  Gtk.Main.Init;
<...>
```

Le code source complet de Hello3 est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Hello3

4. Utilisation avec GLADE et Gate3

Glade est un outil graphique de développement d'interfaces utilisateurs graphiques pour la bibliothèque GTK.

Saisir les commandes suivantes dans le Terminal :

```
$ export XDG_DATA_DIRS=$xnadalib/share
$ $xnadalib/bin/glade &
```

Cliquer sur le bouton *Create a new project*.

Dans la fenêtre principale du projet, ajouter une fenêtre applicative en cliquant sur son icône *GTKWindow* depuis les objets *Toplevels* (figure 1-1), dans l'inspecteur des propriétés modifier son identification avec la propriété *ID* (onglet *General*, figure 1-2a), mettre "Window1" (sans espace ni caractères spéciaux, cet identifiant va servir dans le programme Ada), modifier également son titre avec la propriété *Title*, mettre "Essai 1" (onglet *General*, figure 1-2b). Sauvegarder le projet en cliquant sur *Save* (figure 1-3) avec le nom "Essai1" dans le dossier "Essai1" puis cliquer sur *Save*, le fichier "Essai1.glade" est créé.

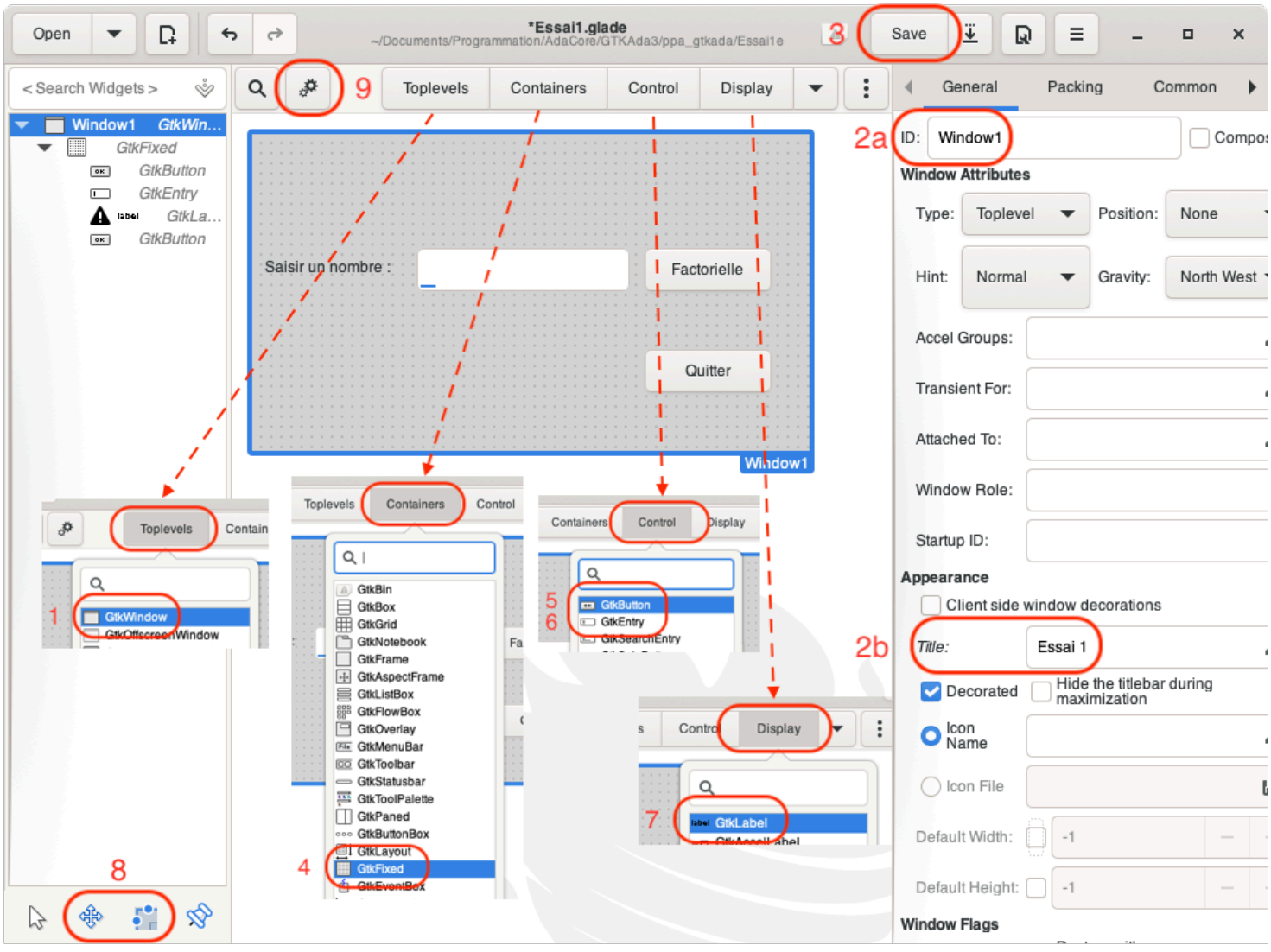


Figure 1 : Construction de l'interface utilisateur avec GLADE

a) Création de l'interface graphique

Reprendre l'exemple ci-dessus avec Glade puis ajouter un objet grille *GTKFixed* depuis les objets *Containers* (figure 1-4) dans la fenêtre (cliquer à l'intérieur) pour avoir une grille de positionnement de nos prochains objets.

Ajouter un bouton *GTKButton* depuis les objets *Control* (figure 1-5), modifier son texte avec le champ *Label with optional image* (inspecteur d'objet dans onglet *General*), mettre "Quitter".

Ajouter une zone de saisie *GTKEntry* depuis les objets *Control* (figure 1-6), dans l'inspecteur des propriétés modifier son identification avec la propriété *ID* (onglet *General*), mettre "entry1" (sans espace ni caractères spéciaux, cet identifiant va servir dans le programme Ada).

Ajouter un affichage *GTKLabel* depuis les objets *Display* (figure 1-7), modifier son texte avec le champ *Label* (inspecteur d'objet onglet *General*), mettre "Saisir un nombre :".

Ajouter un bouton *GTKButton* depuis les objets *Control* (figure 1-5), modifier son texte avec le champ *Label with optional image* (inspecteur d'objet onglet *General*), mettre "Factorielle".

Les objets graphiques sont ajustables en position et en taille dans les onglets *Packing* (propriétés *X position* et *Y position*) et *Common* (propriétés *Width request* et *Height request*) ainsi qu'avec la souris en cliquant sur les outils *Drag* et *Resize*" (figure 1-8).

Il est déjà possible d'interagir avec notre interface en sélectionnant la fenêtre et en cliquant sur le bouton *Preview snapshot* (figure 1-9).
Sauvegarder le projet en cliquant sur *Save.*

b) Gestion des évènements

Nous allons attacher le calcul de la factorielle au bouton éponyme. Dans Glade, sélectionner le bouton Factorielle, choisir l'évènement *Clicked* dans *GtkButton* de l'inspecteur d'objet onglet *Signals*. Cliquer sur *<Type here>* dans la colonne *Handler* et saisir "on_button2_clicked".
Sauvegarder, notre fichier Glade peut maintenant être transformé par le programme *gate3* en code source GTKAda.

Nous ajoutons auparavant un projet GPR, Le code source *essai1.gpr* :

```
with "gtkada.gpr";
project Essai1 is
   for Main use ("essai1.adb");
end Essai1;
```

Dans le Terminal, saisir les commandes suivantes pour générer les sources Ada et l'exécutable :

```
$ cd <dossier du projet Essai1>
$ $xnadalib/bin/gate3.sh Essai1.glade
$ gprbuild -P essai1.gpr
$ ./essai1
```

Notre fenêtre s'affiche, les fichiers créés sont :

- essai1.adb : le programme principal de création des fenêtres,
- window1_callbacks.ads : spécification des gestionnaires d'évènements,
- window1_callbacks.adb : code des gestionnaires d'évènements.

Cela doit donner à peu près ceci :

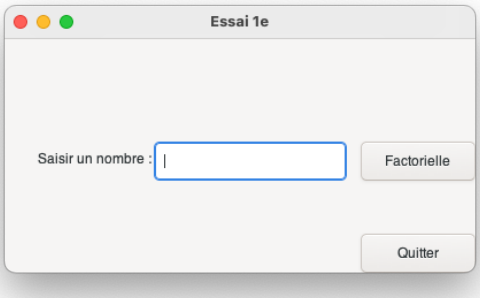


Figure 2 : Notre fenêtre

Arrêter le programme avec *<Ctrl> C*.

Nous allons aussi rendre actif le bouton *Quitter* et la fermeture de la fenêtre.
Sélectionner le bouton *Quitter*, choisir l'évènement *clicked* dans *GtkButton* de l'inspecteur d'objet onglet *Signals*. Cliquer sur *<Type here>* dans la colonne *Handler* et saisir "on_button1_clicked".

Sélectionner la fenêtre principale *window1* (figure 3-1) dans la hiérarchie des objets, choisir l'évènement *delete-event* (figure 3-2) dans *GtkWidget* de l'inspecteur d'objet onglet *Signals* (figure 3-3). Cliquer sur *<Type here>* dans la colonne *Handler* et saisir "on_window1_delete_event".

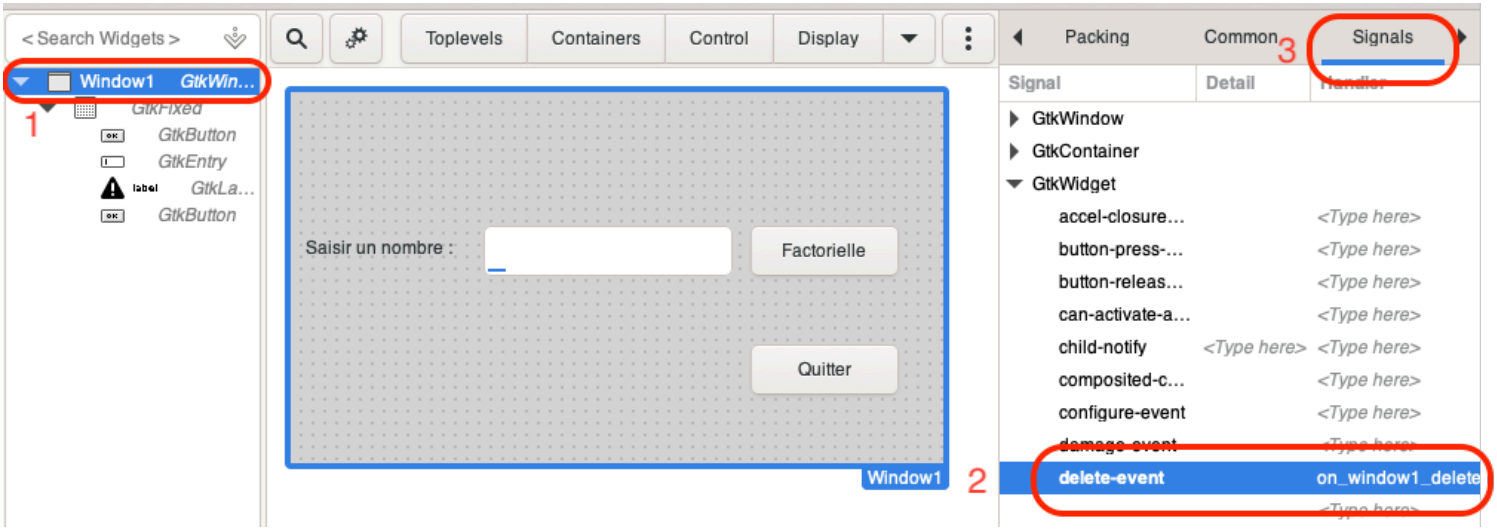


Figure 3 : Propriétés -> Signaux

Sauvegarder et régénérer le code Ada :

$ $xnadalib/bin/gate3.sh Essai1.glade

Nous allons maintenant mettre les mains dans le code en ajoutant une fonction de récupération de l'instance de notre fenêtre pour l'utilisation des procedures Get_Text et Set_Text avec l'objet "Text Entry".

Dans le source *window1_callbacks.adb* remplacer le code de la procédure *On_Button2_Clicked* par :

```
procedure On_Button2_Clicked (Builder : access Gtkada_Builder_Record'Class) is
   S : constant UTF8_String := Get_Text(Gtk_Entry (Get_Object (Builder, "entry1")));
   function Factorielle (N : Natural) return Long_Long_Integer is
      F : Long_Long_Integer := 1;
   begin
      for I in 2 .. N loop
         F := F * Long_Long_Integer(I);
      end loop;
      return F;
   end Factorielle;
begin
   Set_Text(Gtk_Entry (Get_Object (Builder, "entry1")),
      Long_Long_Integer'Image(Factorielle(Natural'Value(S))));
exception
   when others =>
      Set_Text(Gtk_Entry (Get_Object (Builder, "entry1")), "Erreur!");
end On_Button2_Clicked;
```

Ensuite remplacer l'appel à *Put_Line* dans la procédure *On_Button1_Clicked* par :

```
   Gtk.Main.Main_Quit;
```

Et supprimer juste l'appel à *Put_Line* dans la fonction *On_Window1_Delete_Event*.
Puis ajouter les déclarations des bibliothèques avant "with Gtk.Main;" :

```
with Glib;       use Glib;
with Glib.Object;use Glib.Object;
with Gtk.Enums;  use Gtk.Enums;
with Gtk.Widget; use Gtk.Widget;
with Gtk.GEntry; use Gtk.GEntry;
```

Compiler le code et exécuter le programme :
(L'option -gnato permet de déclencher une exception en cas d'erreur numérique)

```
$ gprbuild -gnato -P essai1.gpr
$ ./essai1
```

La fenêtre est alors entièrement active avec le calcul de factorielle.

Le code source complet de Essai1 est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Essai1e

c) Ajout de widgets

Reprenons l'exemple précédent en remplaçant l'objet grille *Fixed* par *Grid* de 5 lignes et 3 colonnes. Placer les 3 widgets précédents sur la deuxième ligne comme en a).

Nous allons ajouter deux étiquettes *GTKLabel* pour afficher le résultat et le texte "décimale(s)", une case à cocher *GTKCheckButton* pour un calcul avec des réels plutôt que des entiers et un curseur *GTKScale* pour déterminer le nombre de décimales à afficher.

Avec Glade, sélectionner tour à tour les quatre widgets depuis les objets *Contrôle* et *Affichage* pour les positionner dans la fenêtre principale et modifier leur affichage comme Figure 4.

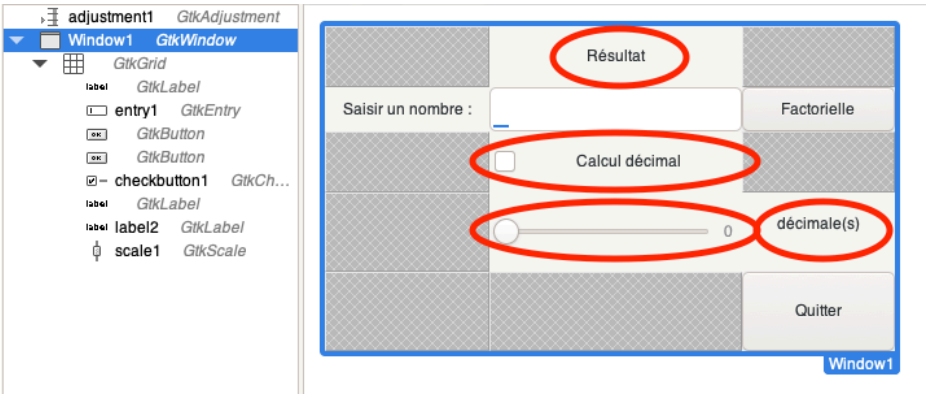


Figure 4 : Les nouveaux widgets

Ajouter les ID suivants (comme indiqué sur la figure 4):

- "entry1" pour la zone de saisie,
- "label2" pour l'étiquette de résultat,
- "scale1" pour le curseur,
- "checkbutton1" pour la case à cocher.

Pour le curseur, il nous faut modifier quelques paramètres supplémentaires dans Glade onglet *General* :

- Digits : 0,
- Value Position : Right,
- Orientation : Horizontal,
- Adjustement : cliquer sur le crayon, une fenêtre qui s'ouvre, cliquer sur New.

Ensuite, sélectionner *adjustment1* dans l'inspecteur de widgets, modifier les propriétés "*General*" comme sur la figure 5 :

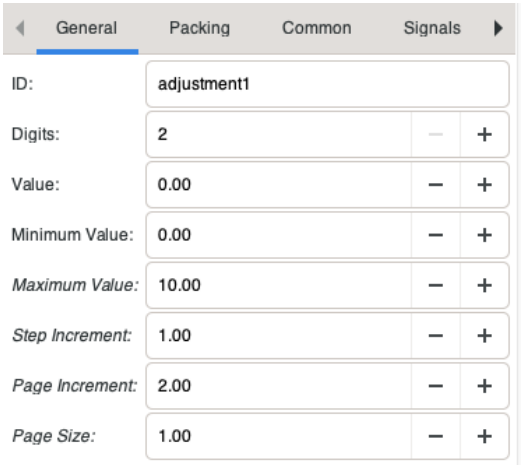


Figure 5 : Propriétés -> General

Nous allons poursuivre en ajoutant les évènements comme en b).

Maintenant, nous avons l'habitude, sauvegarder sous le nom Essai2 (auparavant ne pas oublier de changer de nom la fenêtre avec la propriété "Window Title") et générer le code Ada :

```
$ cd <dossier du projet Essai2>
$ $xnadalib/bin/gate3.sh Essai2.glade
$ gprbuild -P essai2.gpr
$ ./essai2
```

Nous pouvons alors ajouter le code correspondant dans le fichier window1_callbacks.adb :

- fonction On_Window1_Delete_Event : supprimer l'appel à "Put_Line",
- procédure On_Button1_Clicked : remplacer l'appel à "Put_Line" par "Gtk.Main.Main_Quit;"
- procédure On_Button2_Clicked :remplacer le code par :

```
procedure On_Button2_Clicked (Builder : access Gtkada_Builder_Record'Class) is
   S : constant UTF8_String := Get_Text (Gtk_Entry (Get_Object (Builder, "entry1")));
   function Factorielle (N : Natural) return Long_Long_Integer is
      F : Long_Long_Integer := 1;
   begin
      for I in 2 .. N loop
         F := F * Long_Long_Integer (I);
      end loop;
      return F;
   end Factorielle;
   function Factorielle (N : Natural) return String is
      F : Long_Long_Float := 1.0;
      S : String (1 .. 20);
   begin
      for I in 2 .. N loop
         F := F * Long_Long_Float (I);
      end loop;
      Ada.Long_Long_Float_Text_IO.Put
        (S, F,
         Integer (Gtk_Hscale (Get_Object (Builder, "scale1")).Get_Value), 6);
      return S;
   end Factorielle;
begin
   if not Gtk_Check_Button (Get_Object (Builder, "checkbutton1")).Get_Active then
      Set_Text
        (Gtk_Label (Get_Object (Builder, "label2")),
         Long_Long_Integer'Image (Factorielle (Natural'Value (S))));
   else
      Set_Text (Gtk_Label (Get_Object (Builder, "label2")), Factorielle (Natural'Value (S)));
   end if;
exception
   when others =>
      Set_Text (Gtk_Label (Get_Object (Builder, "label2")), "Erreur!");
end On_Button2_Clicked;
```

- ajouter les déclarations des bibliothèques avant "with Gtk.Main;" :

```
with Glib;                    use Glib;
with Gtk.Enums;               use Gtk.Enums;
with Ada.Long_Long_Float_Text_IO;
with Gtk.Scale;       use Gtk.Scale;
with Gtk.Check_Button; use Gtk.Check_Button;
with Gtk.Label;       use Gtk.Label;
with Gtk.GEntry; use Gtk.GEntry;
```

Compiler le code et exécuter le programme :

```
$ gprbuild -gnato -P essai2.gpr
$ ./essai2
```

La fenêtre est alors entièrement active. Le code source complet de Essai12 est disponible ici : github.com/Blady-Com/ppa_gtkada/tree/master/Essai2d

5. Cairo : graphiques en 2D

Cairo est intégré à GTK en tant que support pour les graphiques en 2D mais peut aussi s'utiliser seul. Il s'agit aussi d'un concept de création graphique qui demande de bien définir quelques mots de vocabulaire. Le premier est vectoriel, concept proche des mathématiques à base de cordonnées et de distances, pour représenter le fonctionnement de Cairo par opposition à bitmap collection de pixels allumés ou éteins à une position donnée. Le mode vectoriel est économe en ressource mémoire et permet des transformations du tracé sans perte de qualité alors que le mode bitmap est plus proche du fonctionnement de l'écran de notre ordinateur. Voyons les autres définitions.

Cairo définit tout d'abord trois niveaux de claque et leur structure de données :

- *Destination* : est la surface sur laquelle le tracé va s'appliquer, elle est habituellement vierge,
- *Source* : est la surface à partir de laquelle le tracé va être pris, elle est habituellement complètement noire mais peut être un motif ou une précédente destination,
- *Mask* : est la surface qui filtre le tracé, habituellement elle autorise tout le tracé.
- *Context* : est la structure de données qui enregistre une destination, une source, un mask, le style du tracé et le tracé lui même. Il est créé et détruit par :
  - *Create* : crée un context avec la surface destination en paramètre
  - *Destroy* : détruit le context

Cairo définit alors les procédures permettant de fixer le tracé sur la destination à partir de la source et à travers le mask :

- *Stroke* : transfert un tracé "fil de fer",
- *Fill* : transfert un tracé plein,
- *Show_Text* : transfert un texte.
- *Paint* : transfert la totalité de la source (sans mask)
- *Mask* : transfert la source à travers un masque externe

Le tracé est alors perdu, avec les procédures *Stroke_Preserve* et *Fill_Preserve* le tracé est conservé par le context.

L'application de la source est conditionné par :

- *Set_Operator* : opérateur de composition avec le paramètre prenant une valeur parmi : (beaucoup d'autres sont définis)
  - *Cairo_Operator_Clear* : efface la destination correspondant au tracé
  - *Cairo_Operator_Source* : remplace la destination par le tracé
  - *Cairo_Operator_Over* : applique le tracé sur la destination avec composition des couleurs (mode par défaut)
- *Set_Source* : affecte la source par le motif en paramètre
- *Set_Source_RGB* : modifie la couleur de la source
- *Set_Source_Surface* : affecte la source par la surface en paramètre

Les procédures de création d'un tracé sont plus communes :

- *Move_To* : place le point courant aux coordonnées en paramètres
- *Line_To* et *Line_Rel* : trace une ligne droite vers un point ou d'une distance donnés en paramètre
- *Arc* : trace un arc de cercle avec le centre, le rayon et les angles de départ et d'arrivée en paramètres
- *Rectangle* : trace un rectangle avec le point en haut à gauche, la longueur et la largeur en paramètre
- *New_Sub_Path* : commence un nouveau tracé sans affecter le précédent (utile notamment avec Arc qui établit un lien avec le précédent tracé)

Le système de coordonnées est indépendant du dispositif d'affichage. Ainsi, il reprend quelques concepts mathématiques :

- l'axe des abscisses (X ou horizontal) croissant va de la gauche vers la droite,
- l'axe des ordonnées (Y ou vertical) croissant va de haut en bas (au contraire de l'habitude en mathématiques),
- les coordonnées sont des nombres réels,
- les deux axes se coupent à l'origine (0.0, 0.0),
- les angles croissants vont de l'axe X positif vers l'axe Y positif soit dans le sens horaire (au contraire de l'habitude en mathématiques). Ils sont donnés en radians soit l'angle en degrés * PI / 180.0.

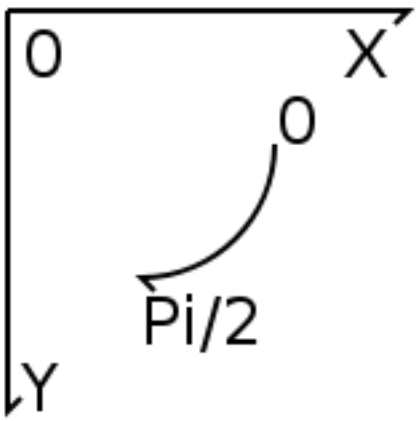


Figure 6 : Le système de coordonnées de Cairo

Le context contient également une matrice de transformation par défaut qu'il est possible de modifier par :

- *Translate* : modifie la matrice par translation de l'origine avec les valeurs de distance en paramètre
- *Rotate* : modifie la matrice par rotation des axes avec la valeur d'angle en paramètre
- *Scale* : modifie la matrice par dilatation de l'échelle avec les valeurs de facteurs en paramètre (modifie également l'épaisseur des traits)
- *Identity_Matrix* : revient aux valeurs initiales
- *Save* : sauvegarde le context courant dans une pile
- *Restore* : restore le context courant depuis la pile

Les procédures de création d'un texte sont simples mais rudimentaires. Bien qu'il soit préférable d'utiliser Pango (voir paragraphe à venir), quelques procédures sont bien utiles :

- *Select_Font_Face* : sélectionne une police de caractère
- *Set_Font_Size* : définit la taille des caractères
- *Show_Text* : affiche le texte en paramètre

Il ne nous reste plus qu'à créer la surface destination pour pouvoir l'afficher.

a) Avec Cairo.Image_Surface 1, 8, 24 ou 32 bits

Une fonction pour chaque format va faire l'affaire :

- *Create* : créée une surface avec en paramètre le type de format des pixels et la taille du graphique (largeur et hauteur en pixel)
- *Cairo_Format_ARGB32* : 32 bits (alpha, rouge, vert et bleu sur 8 bits)
- *Cairo_Format_RGB24* : 24 bits (rouge, vert et bleu sur 8 bits)
- *Cairo_Format_A8* : 8 bits pour alpha
- *Cairo_Format_A1* : 1 bit pour alpha

Opérations :

- Création de la surface 1, 8, 24 ou 32 bits
- Création du context
- Dessin avec le context
- Destruction du context
- Destruction de la surface

b) Avec PNG

Une surface est créer avec la fonction du a) puis enregistrée à la fin du tracé avec :

- *Write_To_Png* : enregistre la surface dans un fichier PNG avec en paramètre le nom du fichier

Opérations :

- Création de la surface 1, 8, 24 ou 32 bits
- Création du context
- Dessin avec le context
- Enregistrement de la surface
- Destruction du context
- Destruction de la surface

c) Avec Cairo.PDF et Cairo.SVG

Une fonction pour chaque format va faire l'affaire :

- *Create* : créée une surface PDF ou SVG avec en paramètre le nom du fichier et la taille du graphique (largeur et hauteur en pixel)

Opérations :

- Création de la surface PDF ou SVG
- Création du context
- Dessin avec le context

- Destruction du context
- Transfert de la surface dans le fichier
- Destruction de la surface

d) Avec GTK

Opérations dans le programme principal :

- Création fenêtre *GTK_Window*
- Création zone graphique *GTK_Drawing_Area* attaché à la fenêtre
- Connexion de la procédure de dessin à l'évènement de rafraichissement à la zone graphique
- Opérations dans la procédure de dessin :
- Création d'un context avec la fenêtre de la zone graphique en paramètre
- Dessin avec le context
- Destruction du context

e) Avec Cairo.Surface

Procédures de transfert et destruction de la surface créée précédemment :

- *Show_Page* : transfert et efface la page courante
- *Copy_Page* : transfert la page courante sans l'effacer
- *Destroy* : détruit la surface en paramètre

f) Un exemple

Le programme dessine la figure 6 et la sauvegarde sous les formats PDF, PNG et SVG.
Le code source complet de Hello_Cairo est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Hello_Cairo

Saisir les commandes suivantes dans le Terminal :

```
$ cd <dossier du projet Essai_Cairo>
$ gprbuild -P hello_cairo.gpr
$ ./hello_cairo
```

6. Le multitâche

Avec GTK le programme principal est minime se réduisant presque aux appels à GTK.Main.Init et GTK.Main.Main. Le contrôle du programme est donné à GTK par l'appel à GTK.Main.Main. Les seuls contrôles de l'utilisateur sont alors réalisés par des gestionnaires (handlers) déclenchés par des évènements. Pour permettre d'autres contrôles de type asynchrone, notamment avec de tâches indépendantes, il nous faut prendre quelques précautions pour ne pas perturber GTK.

L'unité GDK.Threads va nous aider avec les procédures d'initialisation : Gdk.Threads.G_Init et Gdk.Threads.Init ainsi que les procédures encadrant les appels à ne pas perturber : Gdk.Threads.Enter et Gdk.Threads.Leave.

Le programme principal de notre application ressemble alors à ceci :

```
begin
   --  Initialize GtkAda with task-safe mode
   Gdk.Threads.G_Init;
   Gdk.Threads.Init;
   Gtk.Main.Init;
...
   --  Start the Gtk+ main loop in task-safe mode
   Gdk.Threads.Enter;
   Gtk.Main.Main;
   Gdk.Threads.Leave;
end;
```

À noter que la protection du traitement des évènements GTK n'est pas requise puisque déjà acquise par GTK.Main.Main que l'on encadre des deux procédure de protection dans le programme principal. Cependant ce n'est pas le cas pour les autres évènements comme les TimeOuts qui faudra encadrer.

Attention également aux tâches Ada qui peuvent démarrer dès la phase d'élaboration, avant donc l'appel à l'initialisation de GDK.Threads et Gtk.Main. Il faudra les bloquer et débloquer par un rendez-vous Ada par exemple.

L'exemple proposé affiche un texte qui se transforme en une suite d'astérisques qui vont se multiplier jusqu'à faire exploser la fenêtre. Appuyer sur le bouton "Diminue" permet de réduire leur nombre de façon asynchrone. Le programme s'arrête lors le nombre d'astérisques dépasse 20.

Le code source complet du programme est disponible ici :
github.com/Blady-Com/ppa_gtkada/tree/master/Tasking

Saisir les commandes suivantes dans le Terminal :

```
$ cd <dossier du projet Tasking>
$ gprbuild -P main.gpr
$ ./main
```

7. Les évènements

Les évènements sont issus de l'interface utilisateur à travers des widgets. Un widget émet un signal comme un message. Les signaux se différencient par leur nom comme "clicked" pour clic sur un bouton. Le principe de base est de connecter une action (un sous-programme) avec un évènement. L'application réagit à ce signal en appellant le sous-programme (handler ou callback) qui y est connecté.

a) Les gestionnaires de handlers

Au sein de l'unité Gtk.Handlers, GtkAda définit six gestionnaires génériques pour connecter un handler :

- Callback : dédié aux handlers sous forme de simple procédure
- User_Callback : dédié aux handlers sous forme de simple procédure et comprenant un paramètre dont le type est défini par l'utilisateur
- User_Callback_With_Setup : identique au précédent en ajoutant la possibilité de définir une procédure d'initialisation du paramètre utilisateur lors de la connexion
- Return_Callback : dédié aux handlers sous la forme d'une fonction renvoyant une valeur
- User_Return_Callback : dédié aux handlers sous la forme d'une fonction renvoyant une valeur et comprenant un paramètre dont le type est défini par l'utilisateur
- User_Return_Callback_With_Setup : identique au précédent en ajoutant la possibilité de définir une procédure d'initialisation du paramètre utilisateur lors de la connexion

Chacun des gestionnaires défini deux handlers : un premier handler pour les signaux accompagnés d'un certain nombre de valeurs du type Glib.Values.GValues et un deuxième plus simple sans valeurs qui peut être aussi utilisé avec des signaux accompagnés de valeurs mais nous n'y aurons alors pas accès. C'est au handler d'utiliser ou non les valeurs accompagnant le signal et d'en faire la bonne interprétation ou conversion. Nous verrons par la suite les Marshallers qui vont nous y aider.

Pour instancier un gestionnaire il faut donc connaitre le type de widget, la nature du handler à connecter (procédure ou fonction) suivant le signal émis, éventuellement le type du paramètre utilisateur et si besoin sa procédure d'initialisation. Ces informations sont issues de la documentation de GTK au paragraphe "Signal Details" de chaque widget. L'instance ne doit pas être créée à l'intérieur d'un bloc de code mais au niveau paquetage pour être actif en mémoire de façon à répondre à un évènement à tout moment.

Exemple avec un bouton :

```
The "clicked" signal
void user_function (GtkButton *button, gpointer user_data);
```

On utilisera :

```
package Button_Callback is new
  Gtk.Handlers.Callback (Gtk_Button_Record);
```

ou :

```
package Button_Callback is new
  Gtk.Handlers.Callback (Gtk_Button_Record, T_User_Data);
```

ou :

```
package Button_Callback is new
  Gtk.Handlers.Callback (Gtk_Button_Record, T_User_Data, Init);
```

Exemple avec une fenêtre :

The "delete-event" signal

```
gboolean user_function (GtkWidget *widget, GdkEvent *event, gpointer user_data);
```

On utilisera :

```
package Window_Callback is new
  Gtk.Handlers.Return_Callback (Gtk_Widget_Record, Boolean);
```

etc.

b) La spécification des handlers

Le handler prends alors une forme avec plusieurs paramètres définis par le gestionnaire instancié et le type signal émis (avec valeurs ou non) :

- le widget auquel le handler a été connecté :

```
procedure Handler (Widget : access Gtk_Widget_Record'Class);
```

- auquel peut s'ajouter une valeur comme un évènement :

```
procedure Handler
  (Widget : access Gtk_Widget_Record'Class;
   Event  : Gdk.Event.Gdk_Event);
```

- auquel peut s'ajouter des données utilisateurs :

```
procedure Handler
  (Widget    : access Gtk_Widget_Record'Class;
   Event     : Gdk.Event.Gdk_Event;
   User_Data : T_User_Data);
```

- avec une valeur à retourner :

```
function Handler (Window : access Gtk_Window_Record'Class;
                  Event : Gdk.Event.Gdk_Event)
          return Boolean;
```

etc.

Vous remarquerez que ces exemples de handler ne correspondent pas toujours avec les prototypes de handlers définis par défaut dans les gestionnaires GTKAda.

En fait, nous pouvons utiliser des handlers directement définis par défaut dans le gestionnaire avec des valeurs de type générique GValues où il nous faudra faire les conversions manuellement. Ou bien nous pouvons écrire des handlers avec des paramètres de types plus élaborés convertis automatiquement grâce à des convertisseurs génériques (Marshaller) définis dans les gestionnaires de handlers.

Ils sont au nombre de 6 à 8 suivant les gestionnaires :

- Marshallers : pas de paramètre
- Gint_Marshaller : convertit le premier paramètre en Gint
- Guint_Marshaller : convertit le premier paramètre en Guint
- Event_Marshaller : convertit le premier paramètre en Gdk_Event
- Widget_Marshaller : convertit le premier paramètre en Gtk_Widget_Record
- ...
- Tree_Path_Tree_Iter_Marshaller : convertit le premier paramètre en Gtk_Tree_Path et le second en Gtk_Tree_Iter

Suivant le signal connecté nous définissons un handler correspondant déterminé là aussi dans la documentation de GTK au paragraphe "Signal Details" de chaque widget. Le choix du handlers se fera automatiquement au moment de la connexion par le gestionnaire de handler, par exemple :

The "button-press-event" signal

```
gboolean user_function (GtkWidget *widget,
                        GdkEvent  *event,
                        gpointer   user_data);
```

On écrira :

```
function Handler (Widget : access Gtk_Widget_Record'Class;
                  Event : Gdk.Event.Gdk_Event;
                  User_Data : TUser_Data)
         return Boolean;
```

c) La connexion

Nous connectons alors le widget émetteur, le signal émis avec le handler définis avec Marshaller ou non.

Des nom de signaux prédéfinis apparaissent dans chaque unité définissant chaque widget, par exemple Gtk.Button définit :

```
Signal_Activate : constant Glib.Signal_Name := "activate";
Signal_Clicked : constant Glib.Signal_Name := "clicked";
Signal_Enter : constant Glib.Signal_Name := "enter";
Signal_Leave : constant Glib.Signal_Name := "leave";
Signal_Pressed : constant Glib.Signal_Name := "pressed";
Signal_Released : constant Glib.Signal_Name := "released";
```

Plusieurs handlers peuvent être connectés au même signal, ils seront appelés dans l'ordre de leur connexion sauf si le paramètre optionnel After est positionné à True (il est False par défaut).

Exemple d'un bouton connecté à un handler avec Marshaller dans l'ordre des connexions pour le signal "clicked" :

```
Button_Callback.Connect
  (Button1, Signal_Clicked, To_Marshaller (On_Button1_Clicked'Access), False);
```

Suivant l'appel avec Marshaller ou pas, la procédure "Connect" correspondante sera appelée.
Les connexions seront détruites automatiquement lors de la destruction du widget.
Une valeur peut être retournée par la connexion pour pouvoir être réutilisée plus tard pour réaliser une déconnexion par exemple :

```
Id : Gtk.Handlers.Handler_Id;
-- ...
Id := Return_Callback.Connect
  (Window1, Signal_Delete_Event, On_Window1_Delete_Event'Access, True);
-- ...
Gtk.Handlers.Disconnect(Window1, Id);
```

Les signaux doivent également être autorisés par le widget connecté au moyen de Set_Events avec pour paramètre le masque des évènements autorisés. Cette information est issue de la documentation de GTK au paragraphe "Signal Details" de chaque widget :

Exemple :
The "button-press-event" signal
...
To receive this signal, the GdkWindow associated to the widget needs to enable the GDK_BUTTON_PRESS_MASK mask.

```
Gtk.Widget.Set_Events
  (Gtk.Widget.Gtk_Widget (Area_Draw),
   Gdk.Event.Button_Press_Mask or Gdk.Event.Button_Release_Mask);
```

d) Le code des handlers

Retournons une nouvelle fois dans la documentation de GTK au paragraphe "Signal Details" de chaque widget pour déterminer l'utilisation des paramètres et la valeur de retour à donner, s'il y en a une. Par exemple pour le signal "button-press-event", la valeur de retour est True pour stopper ou False pour continuer le traitement du signal par les handlers connectés ou celui par défaut :

The "button-press-event" signal
...
widget : the object which received the signal.
event : the GdkEventButton which triggered this signal.
user_data : user data set when the signal handler was connected.
Returns : TRUE to stop other handlers from being invoked for the event. FALSE to propagate the event further.

Certains signaux appellent les handlers utilisateurs avant ou après le handler par défaut suivant les indications de la documentation :

- RUN_FIRST (Handler par défault avant),

- RUN_LAST (Handler par défault après),
- RUN_BOTH (Handler par défault avant et après),
- RUN_ACTION (pas d'action particulière).

Un handler peut servir à traiter plusieurs type de signaux. Dans le code du handlers, nous pouvons utiliser l'évènement en paramètre avec les fonctions du package Gdk.Event, en voici les principales :

- Get_Event_Type : retourne le type d'évènement (souris, clavier,...)
- Get_Window : retourne la fenêtre ayant généré l'évènement
- Get_Time : retourne le temps d'occurence
- Get_Button : retourne le numéro du bouton de la souris
- Get_State : retourne l'état des boutons de la souris et des touches spéciales du clavier
- Get_Key_Val : retourne le code de la touche actionnée

Attention une erreur se produira en utilisant une fonction ne correspondant pas au type attendu de l'évènement, voir les champs pertinents dans la documentation de GDK.Event à chaque type d'évènement comme Gdk_Event_Motion par exemple.

Les états des boutons de la souris et des touches spéciales du clavier sont testés avec les constantes correspondantes ci-dessous :
(l'état lu est celui juste avant l'évènement courant)

- GDK_SHIFT_MASK : majuscule
- GDK_LOCK_MASK : majuscule bloquée
- GDK_CONTROL_MASK : contrôle
- GDK_MOD2_MASK : commande ou pomme
- GDK_RELEASE_MASK : option ou alternate
- GDK_BUTTON1_MASK : bouton gauche
- GDK_BUTTON2_MASK : bouton du milieu
- GDK_BUTTON3_MASK: bouton droit

En annexe A : Un exemple de handler pour des évènements du clavier.
En annexe B : Un exemple de handler pour des évènements de la souris y compris la mollette.

Pascal Pignard, avril-décembre 2011, janvier-décembre 2012, décembre 2013, décembre 2015, juillet 2021.

A. Annexe - Exemple de handler du clavier

```
function On_Key_Press_Event
  (Object : access Gtk.Widget.Gtk_Widget_Record'Class;
   Event  : Gdk.Event.Gdk_Event) return Boolean
is
   pragma Unreferenced (Object);

   function Get_String (Event : Gdk.Event.Gdk_Event) return String is
      Event_Type : constant Gdk.Event.Gdk_Event_Type := Gdk.Event.Get_Event_Type (Event);
      use type Gdk.Event.Gdk_Event_Type, Gtkada.Types.Chars_Ptr;
   begin
      if Event_Type = Gdk.Event.Key_Press or else Event_Type = Gdk.Event.Key_Release then
         declare
            Str : constant Gtkada.Types.Chars_Ptr := Event.Key.String;
         begin
            if Str = Gtkada.Types.Null_Ptr then
               return "";
            end if;
            return Gtkada.Types.Value (Str);
         end;
      end if;
      raise Constraint_Error;
   end Get_String;

   Key : constant Gdk.Types.Gdk_Key_Type := Gdk.Event.Get_Key_Val (Event);
   Ch  : constant String                 := Get_String (Event);
   use Gdk.Types.Keysyms;
   use type Gdk.Types.Gdk_Key_Type;
   use type Gdk.Types.Gdk_Modifier_Type;

begin
   if Ch'Length > 1 then -- UTF8 char
      Write_Key (Ada.Strings.UTF_Encoding.Strings.Decode (Ch) (1));
      return True;
   elsif Ch'Length /= 0 then -- ASCII char
      -- Ctrl-C
      if Char'Pos (Ch (1)) = Char'Pos ('C') - Char'Pos ('@') then
         Gtk.Text_Buffer.Copy_Clipboard
           (Gtk.Text_View.Get_Buffer (Aera_Text),
            Gtk.Clipboard.Get);
         return True;
      end if;
      -- Ctrl-V
      if Char'Pos (Ch (1)) = Char'Pos ('V') - Char'Pos ('@') then
         declare
            S : constant String :=
              Ada.Strings.UTF_Encoding.Strings.Decode
                (Gtk.Clipboard.Wait_For_Text (Gtk.Clipboard.Get));
         begin
            for Ind in S'Range loop
               Write_Key (S (Ind));
            end loop;
         end;
         return True;
      end if;
```

```
      Write_Key (Ch (1));
      return True;
   end if;
  -- Other special keys
   case Key is
      when GDK_BackSpace =>
         Write_Key (Ada.Characters.Latin_1.BS);
      when GDK_Tab =>
         Write_Key (Ada.Characters.Latin_1.HT);
      when GDK_F1 .. GDK_F10 =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         -- Alt modifier
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Release_Mask) /= 0 then
            Write_Key (Char'Val (Key - GDK_F1 + 104));
         -- Control modifier
         elsif (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key (Char'Val (Key - GDK_F1 + 94));
         -- Shift modifier
         elsif (Gdk.Event.Get_State (Event) and Gdk.Types.Shift_Mask) /= 0 then
            Write_Key (Char'Val (Key - GDK_F1 + 84));
         -- No modifier
         else
            Write_Key (Char'Val (Key - GDK_F1 + 59));
         end if;
      when GDK_Home =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key ('w');
         else
            Write_Key ('G');
         end if;
      when GDK_Left =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key ('s');
         else
            Write_Key ('K');
         end if;
      when GDK_Up =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key (Char'Val (141));
         else
            Write_Key ('H');
         end if;
      when GDK_Right =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key ('t');
         else
            Write_Key ('M');
         end if;
      when GDK_Down =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key (Char'Val (145));
```

```
         else
            Write_Key ('P');
         end if;
      when GDK_Page_Up =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key (Char'Val (132));
         else
            Write_Key ('I');
         end if;
      when GDK_Page_Down =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key ('v');
         else
            Write_Key ('Q');
         end if;
      when GDK_End =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key ('u');
         else
            Write_Key ('O');
         end if;
      when GDK_Delete =>
         Write_Key (Ada.Characters.Latin_1.NUL);
         if (Gdk.Event.Get_State (Event) and Gdk.Types.Control_Mask) /= 0 then
            Write_Key (Char'Val (147));
         else
            Write_Key ('S');
         end if;
      when others =>
         null;
   end case;
   return True;
end On_Key_Press_Event;
```

B. Annexe - Exemple de handler de la souris y compris la mollette

```
GTKMaxButton : constant := 5;
type ButtonState is record
   Pressed, Released, DoubleClic, TripleClic         : Boolean;
   Press_Count, Release_Count                        : Integer;
   LastXPress, LastYPress, LastXRelease, LastYRelease : Glib.Gdouble;
end record;
type ButtonsState is array (1 .. GTKMaxButton) of ButtonState;
IntButtons               : ButtonsState;

function On_Mouse_Event
  (Object : access Gtk.Widget.Gtk_Widget_Record'Class;
   Event  : Gdk.Event.Gdk_Event)
   return   Boolean is
   pragma Unreferenced (Object);
   use type Gdk.Event.Gdk_Event_Type;
   LButtonNum : Integer;
   Dir        : Gdk.Event.Gdk_Scroll_Direction;
begin
   if Gdk.Event.Get_Event_Type (Event) = Gdk.Event.Button_Press then
      LButtonNum                    := Integer (Gdk.Event.Get_Button (Event));
      IntButtons (LButtonNum).Pressed := True;
      Gdk.Event.Get_Coords
        (Event,
         IntButtons (LButtonNum).LastXPress,
         IntButtons (LButtonNum).LastYPress);
      if IntButtons (LButtonNum).Press_Count < Integer'Last then
         IntButtons (LButtonNum).Press_Count := IntButtons (LButtonNum).Press_Count + 1;
      else
         IntButtons (LButtonNum).Press_Count := 0;
      end if;
   end if;
   if Gdk.Event.Get_Event_Type (Event) = Gdk.Event.Button_Release then
      LButtonNum                    := Integer (Gdk.Event.Get_Button (Event));
      IntButtons (LButtonNum).Released := True;
      Gdk.Event.Get_Coords
        (Event,
         IntButtons (LButtonNum).LastXRelease,
         IntButtons (LButtonNum).LastYRelease);
      if IntButtons (LButtonNum).Release_Count < Integer'Last then
         IntButtons (LButtonNum).Release_Count := IntButtons (LButtonNum).Release_Count + 1;
      else
         IntButtons (LButtonNum).Release_Count := 0;
      end if;
   end if;
   if Gdk.Event.Get_Event_Type (Event) = Gdk.Event.Gdk_2button_Press then
      IntButtons (Integer (Gdk.Event.Get_Button (Event))).DoubleClic   := True;
   end if;
   if Gdk.Event.Get_Event_Type (Event) = Gdk.Event.Gdk_3button_Press then
      IntButtons (Integer (Gdk.Event.Get_Button (Event))).TripleClic   := True;
   end if;
   if Gdk.Event.Get_Event_Type (Event) = Gdk.Event.Scroll then
      Gdk.Event.Get_Scroll_Direction (Event, Dir);
      IntScroll := To_Integer (Dir);
   end if;
```

```
   return False;
end On_Mouse_Event;
```