GTK+ par l'exemple

par Nicolas Joseph (home) (Blog)

Date de publication : 28 juin 2006

Ce tutoriel a pour but de vous guider dans la réalisation d'une interface graphique en C grâce à GTK+ au travers l'exemple de la réalisation d'un éditeur de texte.

I - Introduction

I-A - But de ce tutoriel

Pour vous initier aux joies de la programmation d'interfaces graphiques, plutôt que de faire un catalogue de toutes les fonctions de la bibliothèque, ce qui serait long et vite ennuyeux, je vous propose de construire pas à pas une application : un éditeur de texte. Pas un simple éditeur de texte, histoire de faire le tour des possibilités de **GTK+**, nous allons y ajouter pas à pas les fonctionnalités suivantes :

- Création, ouverture, enregistrement de fichiers textes
- Possibilité d'ouvrir plusieurs fichiers grâce à une navigation par onglets
- Ouverture rapide d'un fichier grâce à une liste de fichiers.

Rien d'impressionnant par rapport à un traitement de texte mais plus évolué et plus pratique qu'un simple éditeur de texte. Ce tutoriel est organisé de façon à ajouter une nouvelle fonctionnalité à chaque étape tout en apprenant à maîtriser un ou plusieurs éléments de **GTK+**.

Comme je viens de le dire, il n'est pas question de faire un catalogue de la bibliothèque, cependant avant d'utiliser un nouveaux type de widget, j'en ferai une brève description qui pourra vous servir de pense bête lors de vos futurs développements. Cette description comportera un aperçu pour les widgets graphiques, sa hiérarchie d'héritage, ses fonctions de créations (constructeurs), les fonctions les plus utilisées et les signaux importants de la classe.

I-B - Connaissances requises

Le but de ce tutorial est d'apprendre à maîtriser **GTK+** en partant de zéro, ou presque, en effet avant de vouloir créer une interface graphique, il vaut mieux bien connaître le langage C en mode console. Pour cela je vous renvoie au **cours de la rubrique C**.

I-C - Quelques mots sur GTK+

I-C-1 - Historique

GTK+ ou the **G**IMP **T**ool**k**it était à l'origine une boîte à outils pour les développeurs du logiciel **the GIMP** (the **G**NU **I**mage **M**anipulation **P**rogram), qui comme son nom l'indique est un logiciel de manipulation d'images rattaché au projet **GNU**. Au vu de l'importance de cette boîte à outils, **GTK+** a été détaché de the GIMP en septembre 1997. Depuis il existe deux versions : **GTK+** 1.0 et **GTK+** 2.0, versions qui ne sont pas totalement compatibles cependant la première est encore utilisée dans certaines applications, tel que dans le domaine de l'embarqué du fait de sa complexité moindre. Il est bien évident que c'est la seconde version qui est la plus utilisée, elle est à l'origine de nombreux projets tel que le gestionnaire de fenêtre **GNOME** (**G**NU **N**etwork **O**bject **M**odel **E**nvironment). La dernière version en date est la 2.8.1 annoncée le 24 Août 2005.

I-C-2 - Structure

Comme précisé précédemment, **GTK+** est une boîte à outils et, en tant que tel, elle est constituée de plusieurs bibliothèques indépendantes développées par l'équipe de **GTK+** :

- La **GLib** propose un ensemble de fonctions qui couvrent des domaines aussi vastes que les structures de données, la gestion des threads et des processus de façon portable ou encore un analyseur syntaxique pour les fichiers XML et bien d'autre
- **Pango** : il s'agit de la bibliothèque d'affichage et de rendue de textes
- **ATK**.

La bibliothèque **GTK+** en elle-même utilise une approche orientée objet et repose sur plusieurs couches :

- **GObject** : il s'agit de la base de l'implémentation des objets pour la POO
- **GDK** : bibliothèque graphique de bas niveau
- **GTK+** : la bibliothèque **GTK+** elle-même basée sur l'utilisation de widgets (1) .

C'est bien sûr cette dernière qui nous intéresse ici, mais la **glib** nous sera aussi d'une grande aide.

I-C-3 - Pourquoi utiliser GTK+ ?

Effectivement, pourquoi choisir d'utiliser **GTK+** plutôt qu'une autre bibliothèque pour réaliser une interface graphique. Voici quelques arguments :

- **GTK+** est sous licence libre LGPL, vous pouvez donc l'utiliser pour développer des programmes libres ou commerciaux
- La portabilité : **GTK+** est disponible sur un grand nombre de systèmes dont Windows, Linux, Unix et MacOSX
- **GTK+** est développée en C et pour le langage C, cependant elle est disponible pour d'autres langages tels que Ada, C++ (grâce à gtkmm), Java, Perl, PHP, Python, Ruby ou plus récemment C#.

I-D - Installation

I-D-1 - Windows

Pour pouvoir exécuter une application utilisant **GTK+**, il faut commencer par installer les binaires. Pour faciliter leur installation, il existe un installeur :

*Si vous utilisez the Gimp, utilisez les runtimes proposées sur le site **http://www.gimp-win.org**.*

Pour développer une application, il faut les bibliothèques ainsi que les fichiers d'en-tête disponibles sur **gtk.org**. Pour ceux qui utilisent Code::Blocks, voici la méthode à suivre : **Installation de**

I-D-2 - Linux

Sous Linux, vous disposez sûrement d'un système de paquets. Il vous faudra installer deux paquets, le premier contenant les fichiers permettant d'exécuter des programmes utilisant **GTK+**, le second paquet contient les fichiers nécessaires au développement.

Une fois l'installation réussie, compilez à l'aide de la commande :

```
gcc -Werror -Wall -W -O2 -ansi -pedantic `pkg-config --cflags --libs gtk+-2.0` *.c
```

Tous les codes de ce tutoriel ont été testés sous Linux Debian et Windows XP avec Code::Blocks.

I-E - La philosophie GTK+

Voici quelques choix qui ont été faits par l'équipe de développement de **GTK+** :

- Les noms de fonctions sont de la forme *gtk_type_du_widget_action*, ceci rend les noms des fonctions très explicites, en contre partie ils peuvent être très longs
- Tous les objets manipulés sont des *GtkWidget*, pour que **GTK+** vérifie qu'il s'agit du bon type d'objet lors de l'appel à une fonction, chaque type de *widget* possède une macro de la forme GTK_TYPE_DU_WIDGET à utiliser pour transtyper vos *widgets*
- La gestion des événements qui modifient le comportement de l'application (clique de souris, pression d'une touche du clavier...) se fait grâce à l'utilisation de fonctions de rappel (*callback*). Pour cela, on connecte les *widgets* à des fonctions qui seront appelées lorsque l'événement survient.

I-F - Quelques notions de POO

La POO ou **P**rogrammation **O**rientée **O**bjet est un mode de programmation basée sur des objets. **GTK+** utilisant ce principe (2) , je vais vous en expliquer les principes de bases.

Plutôt que le déroulement du programme soit régie par une suite d'instruction, en POO il est régi par la création, l'interaction et la destruction d'objets, en résumé la vie des objets qui le composent. Un objets est une entité (le parallèle avec la vie courante est simple, un objet pourrait être un ordinateur), en C on pourrait se représenter un objet comme une structure de données.

Un objet est composé de propriétés (ce qui correspond à nos variables) et de méthodes (des fonctions). Pour créer un objet, on fait appel à son constructeur, il s'agit d'une fonction qui a pour but de créer et d'initialiser l'objet. A la fin de sa vie, l'objet doit être détruit, c'est le but du destructeur.

Dans la vie courante, on peut créer des objets complexes à partir de briques de base, il est possible de créer de nouveaux objets en se basant sur un ou plusieurs objets : ce mécanisme est appeler héritage (on parle d'héritage multiple lorsqu'il met en jeu plusieurs objets parents). L'objet ainsi obtenu hérite des propriétés et des méthodes de ses parents tout en pouvant modifier leur comportement et créer ses propres propriétés et méthodes.

Lorsque l'on souhaite créer un patron pour de futures classes, plutôt que de créer une classe de base qui ne sera jamais instanciée, il est possible de créer une interface.

Le dernier concept de POO qui va nous servir dans ce tutoriel est le polymorphisme. Ceci permet à un objet (prenons par exemple camion), de se comporter comme l'un de ses parents (dans le cas du camion, il peut être vu comme un véhicule, au même titre qu'une voiture). Ceci est aussi valable pour des classes implémentant la même interface.

Pour vous initier aux joies de la programmation orientée objet en C, je vous conseille le tutoriel de CGI : **POO en C**

I-G - Notre premier programme

Un programme qui utilise **GTK+** est un programme écrit en C avant tout, il contient donc le code de base de tout programme C :

main.c
```
#include <stdlib.h>

int main (int argc, char **argv)
{
  /* ... */
  return EXIT_SUCCESS;
}
```

Pour pouvoir utiliser les fonctions de **GTK+**, il faut bien sûr inclure le fichier d'en-tête correspondant :

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

La première chose à faire est d'initialiser la machinerie **GTK+** grâce à la fonction *gtk_init* :

```
void gtk_init (int *argc, char ***argv);
```

Cette fonction reçoit les arguments passés en ligne de commande, ceux qui sont spécifiques à **GTK+** sont ensuite retirés de la liste; d'où l'utilisation des pointeurs.

Ensuite, il nous faut créer tous les *widgets* dont nous avons besoin, si nécessaire modifier leurs paramètres par défaut, les connecter à des fonctions *callback* et ensuite demander leur affichage (tout ceci sera explicité dans la suite du tutoriel). Une fois la fenêtre principale créée, il suffit de lancer la boucle principale de **GTK+** :

```
void gtk_main (void);
```

Cette fonction est une boucle sans fin (seule la fonction *gtk_main_quit* permet d'en sortir) qui se charge de gérer le déroulement de notre programme (affichage des *widgets*, envoi de signaux...).

Voici donc notre premier programme en C/**GTK+** qui ne fait rien :

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

int main (int argc, char **argv)
{
  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */

  /* Lancement de la boucle principale */
  gtk_main();
  return EXIT_SUCCESS;
}
```

En plus de ne rien faire notre programme ne peut être terminé que de façon brutale (Ctrl+C) puisque nous n'appelons pas la fonction *gtk_main_quit*.

I-H - Code source

chapitre1.zip

II - Notre première fenêtre

II-A - Aperçu

II-B - Création

La fenêtre principale est représentée par la classe *GtkWindow*.

La création d'une fenêtre se fait simplement en appelant le constructeur de cette classe, la fonction *gtk_window_new* :

```
GtkWidget *gtk_window_new (GtkWindowType type);
```

Le seul paramètre de cette fonction est le type de fenêtre souhaité. Il n'y a que deux possibilités :

- GTK_WINDOW_TOPLEVEL : une fenêtre classique (c'est la base de notre application)
- GTK_WINDOW_POPUP : il s'agit d'une fenêtre avec seulement l'espace de travail (pas de bordure ni de menu système).

Cette fonction renvoie un pointeur sur une structure de type *GtkWindow* (transformer en *GtkWidget* grâce au polymorphisme) qui est l'entité qui va nous servir à manipuler notre fenêtre.

II-C - Affichage

Si vous essayez d'écrire un code maintenant, vous ne verrez rien, pourquoi ? Tout simplement parce qu'il faut préciser à **GTK+** qu'il faut rendre notre fenêtre visible grâce à la fonction *gtk_widget_show* :

```
void gtk_widget_show (GtkWidget *widget);
```

Voici le code qui permet d'afficher notre première fenêtre :

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

int main (int argc, char **argv)
{
  GtkWidget *p_window = NULL;

  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */
  p_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  /* Affichage de la fenetre principale */
  gtk_widget_show (p_window);
  /* Lancement de la boucle principale */
  gtk_main ();
  return EXIT_SUCCESS;
}
```

Ceux qui se sont essayés à la création d'applications graphiques avec l'**API Windows** comprendront la simplicité de ce code.

II-D - Destruction

Dans l'exemple précédent, si vous avez essayé de terminer l'application en fermant la fenêtre (Alt+F4 ou en cliquant sur la petite croix), vous vous êtes peut-être aperçu que l'application tournait encore en fond, c'est le même problème que pour le chapitre précédent : on ne fait pas appel à *gtk_main_quit*. Mais comment appeler cette fonction puisque qu'une fois *gtk_main* appelée nous ne pouvons rien faire ? C'est là qu'intervient le mécanisme des *callback*. A chaque événement qui se produit, la bibliothèque **GObject** produit un signal, si nous souhaitons modifier le comportement par défaut du signal, il suffit de connecter notre fonction *callback* à l'événement souhaité :

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

Cette macro permet d'intercepter l'événement *detailed_signal* de l'objet *instance* grâce à la fonction *c_handler* qui doit être du type *GCallback* :

```
void (*GCallback) (void);
```

Les fonctions *callback* peuvent bien sûr recevoir des paramètres mais leur nature et leur nombre dépendent du contexte, tout est géré par la bibliothèque **GObject**. Le prototype le plus courant pour une fonction de rappel est le suivant :

```
void callback (GtkWidget *p_widget, gpointer *user_data);
```

Dont le premier paramètre est le *widget* qui a reçu le signal et le second correspond au paramètre *data* de la fonction *g_signal_connect*, qui nous permet de passer des informations aux fonctions *callback*. Pour finir proprement notre application, il suffit d'intercepter le signal *destroy* de notre fenêtre et d'y assigner la fonction *gtk_main_quit* qui ne prend pas d'argument. Voici donc notre première application fonctionnelle :

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

int main (int argc, char **argv)
{
  GtkWidget *p_window = NULL;

  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */
  p_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (p_window), "destroy", G_CALLBACK (gtk_main_quit), NULL);

  /* Affichage de la fenetre principale */
  gtk_widget_show (p_window);
  /* Lancement de la boucle principale */
  gtk_main ();
  return EXIT_SUCCESS;
}
```

Il est plus courant de créer notre propre fonction de rappel pour quitter le programme; ceci permet de libérer la mémoire allouée, fermer les fichiers ouverts...

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

void cb_quit (GtkWidget *, gpointer);

int main (int argc, char **argv)
{
  GtkWidget *p_window = NULL;

  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */
  p_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
   g_signal_connect (G_OBJECT (p_window), "destroy", G_CALLBACK (cb_quit), NULL);

  /* Affichage de la fenetre principale */
  gtk_widget_show (p_window);
  /* Lancement de la boucle principale */
  gtk_main ();
  return EXIT_SUCCESS;
}

void cb_quit (GtkWidget *p_widget, gpointer user_data)
{
  gtk_main_quit();

  /* Parametres inutilises */
  (void)p_widget;
  (void)user_data;
}
```

Le préfixe *cb_* permet de différencier les fonctions **c**all**b**ack, qu'il est préférable de regrouper dans un ou plusieurs fichiers séparés.

A la fin de la fonction, je transtype les paramètres inutilisés pour éviter que mon compilateur m'indique des variables non utilisées, c'est le problème avec les fonctions callback, l'API nous impose un prototype.

II-E - Code source

chapitre2.zip

III - Fermer notre fenêtre grâce aux boutons

III-A - Aperçu

III-B - Les boutons poussoir

Maintenant que notre programme se termine proprement, il est plus convivial de proposer à l'utilisateur un bouton pour quitter. Pour créer un bouton poussoir, il existe plusieurs possibilités :

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

La première fonction crée un bouton qui peut servir pour contenir n'importe quel autre *widget* (label, image...). Si vous souhaitez créer un bouton avec du texte dessus, utilisez directement la seconde fonction qui prend en argument le texte à afficher. Si vous souhaitez ajouter un raccourci clavier (accessible avec la touche Alt), la troisième fonction permet d'en spécifier un en faisant précéder une lettre (généralement la première) d'un underscore '_' (si vous souhaitez afficher le caractère '_', il faut en mettre deux). Enfin la dernière fonction permet d'utiliser les *Stock Items* qui sont un ensemble d'éléments prédéfinis par **GTK+** pour les menus et barres d'outils. Le seul paramètre de cette fonction est le nom de l'élément et **GTK+** se charge d'afficher l'icône appropriée ainsi que le texte suivant la langue choisie et un raccourci.

C'est bien sûr cette dernière fonction que nous allons utiliser et ce le plus souvent possible. Voici le code pour créer un tel bouton :

main.c
```
GtkWidget *p_button = NULL;

p_button = gtk_button_new_from_stock (GTK_STOCK_QUIT);
```

Pour obtenir tous les types de stocks items disponibles : **GtkStockItem**

Bien sûr, comme pour la fenêtre si l'on veut voir quelque chose, il faut demander à **GTK+** d'afficher notre *widget* :

main.c
```
gtk_widget_show (p_button);
```

Mais pas seulement ! En effet, il faut préciser à **GTK+** où doit être affiché notre bouton. Pour cela, la classe *GtkWindow* est dérivée de la classe *GtkContainer* qui est, comme son nom le laisse penser, un conteneur pour *widget*. Pour ajouter un *widget*, il suffit de faire appel à la fonction *gtk_container_add* :

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

Le premier paramètre de cette fonction est un *GtkContainer*, or nous souhaitons passer notre fenêtre qui est un *GtkWidget*, pour ne pas avoir de problèmes, il faut utiliser le système de macro offert par **GTK+** pour transtyper notre *GtkWidget* en *GtkContainer* (eh oui en C, le polymorphisme a ses limites) :

main.c
```
gtk_container_add (GTK_CONTAINER (p_window), p_button);
```

Ce mécanisme de transtypage permet à **GTK+** de vérifier que notre *GtkWidget* est bien compatible avec l'utilisation que l'on souhaite en faire. En cas d'échec, **GTK+** nous prévient en affichant un message dans la console :

```
(gtk_bouton.exe:1044): Gtk-CRITICAL **: gtk_container_add: assertion `GTK_IS_CONTAINER (container)'
 failed
```

Pour finir, il faut connecter notre bouton pour appeler notre fonction *cb_quit* lorsque l'utilisateur clic dessus (ce qui correspond à l'événement "clicked") :

```
g_signal_connect (G_OBJECT (p_button), "clicked", G_CALLBACK (cb_quit), NULL);
```

Pour éviter d'appeler la fonction gtk_widget_show pour tous les widgets de notre application, on peut se contenter d'un seul appel à la fonction gtk_widget_show_all qui permet d'afficher tous les widgets contenus dans celui passé à la fonction.

main.c
```
#include <stdlib.h>
#include <gtk/gtk.h>
#include "callback.h"

int main (int argc, char **argv)
{
  GtkWidget *p_window = NULL;

  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */
  p_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (p_window), "destroy", G_CALLBACK (cb_quit), NULL);

  {
    GtkWidget *p_button = NULL;

    p_button = gtk_button_new_from_stock (GTK_STOCK_QUIT);
    gtk_container_add (GTK_CONTAINER (p_window), p_button);
    g_signal_connect (G_OBJECT (p_button), "clicked", G_CALLBACK (cb_quit), NULL);
  }

  /* Affichage de la fenetre principale */
  gtk_widget_show_all (p_window);
  /* Lancement de la boucle principale */
  gtk_main ();
  return EXIT_SUCCESS;
}
```

III-C - Code source

chapitre3.zip

IV - Comment afficher plusieurs widgets

IV-A - Aperçu

IV-B - Le problème

Dans la partie précédente, nous avons réussi à afficher un bouton dans la fenêtre de notre application. Si vous avez essayé d'ajouter un second *widget*, **GTK+** à dû vous répondre poliment :

```
(gtk_box.exe:492) : Gtk-WARNING **: Attempting to add a widget with type
GtkButton to a GtkWindow, but as a GtkBin subclass a GtkWindow can only contain
one widget at a time; it already contains a widget of type GtkButton
```

Les plus anglophiles d'entre vous auront compris que **GTK+** n'accepte pas l'ajout un second *widget* dans notre fenêtre tout simplement parce qu'il y en a déjà un et que la sous classe *GtkBin* (classe mère de notre *GtkWindow*) ne peut contenir qu'un seul *widget*.

IV-C - La solutions

Pour contourner ce problème, il faut commencer par créer un *widget* qui accepte d'en contenir plusieurs autres, pour ensuite y ajouter tout ce dont nous avons besoin. Pour l'instant, nous utiliserons qu'un seul *widget* (il existe trois classes qui permettent ceci), il s'agit des *GtkBox*. Il s'agit de la méthode que j'utilise le plus souvent. Ce n'est peut-être pas la plus simple à maîtriser mais elle extrêmement souple et puissante. Elle consiste à créer une boîte puis à y ajouter les *widgets* les uns après les autres (soit au début soit à la fin). Il existe deux classes héritant de *GtkBox* : les *GtkVBox* qui empilent les *widgets* dans le sens vertical et les *GtkHBox* qui font de même mais dans le sens horizontal. Les fonctions pour manipuler ces deux classes sont les mêmes, seule la fonction pour les créer portent un nom différent :

```
GtkWidget *gtk_vbox_new (gboolean homogeneous, gint spacing);
GtkWidget *gtk_hbox_new (gboolean homogeneous, gint spacing);
```

Le paramètre *homogeneous* permet de réserver pour chaque *widget* une zone de taille identique (zone que le *widget* n'est pas obligé de remplir) et *spacing* permet d'ajouter une bordure en pixels (espacement autour de la *GtkBox*). Ensuite, il suffit d'ajouter les différents widgets grâce aux fonctions :

```
void gtk_box_pack_start (GtkBox *box, GtkWidget *child, gboolean expand, gboolean fill, guint
 padding);
void gtk_box_pack_end (GtkBox *box, GtkWidget *child, gboolean expand, gboolean fill, guint
 padding);
```

Qui ajoutent réciproquement le *widget child* au début et à la fin de *box*. Le paramètre *expand* permet au *widget* d'avoir le plus de place possible (si le paramètre *homogeneous* vaut **TRUE**, cette option a aucun effet). Si plusieurs *widget* ont ce paramètre à **TRUE**, ils se partagent de façon égale l'espace. L'option *fill* permet au *widget* de remplir tout l'espace qui lui ait réservé.

Pour réellement comprendre l'influence de ces paramètres, il faut faire des tests en modifiant un à un chaque paramètre et observer les effets d'un redimentionnement de la fenêtre principale.

Le plus gênant avec cette méthode c'est qu'il faut jongler entre les *GtkHBox* et les *GtkVBox* en les imbriquant pour obtenir le résultat souhaiter : n'hésitez pas à utiliser une feuille et un crayon ;)

Pour en revenir à notre éditeur de texte, nous allons préparer notre application à contenir les futurs *widgets*. Nous utilisons un *GtkVBox* pour l'ensemble des *widgets* (il s'agit de la boîte principale qui pourra contenir d'autres *GtkContainer* selon nos besoins) :

main.c
```c
#include <stdlib.h>
#include <gtk/gtk.h>
#include "callback.h"

int main (int argc, char **argv)
{
  GtkWidget *p_window = NULL;
  GtkWidget *p_main_box = NULL;

  /* Initialisation de GTK+ */
  gtk_init (&argc, &argv);

  /* Creation de la fenetre principale de notre application */
  p_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (p_window), "destroy", G_CALLBACK (cb_quit), NULL);

  /* Creation du conteneur principal */
  p_main_box = gtk_vbox_new (FALSE, 0);
  gtk_container_add (GTK_CONTAINER (p_window), p_main_box);

  /* Creation du bouton "Quitter" */
  {
    GtkWidget *p_button = NULL;

    p_button = gtk_button_new_from_stock (GTK_STOCK_QUIT);
    g_signal_connect (G_OBJECT (p_button), "clicked", G_CALLBACK (cb_quit), NULL);
    gtk_box_pack_start (GTK_BOX (p_main_box), p_button, FALSE, FALSE, 0);
  }

  /* Affichage de la fenetre principale */
  gtk_widget_show_all (p_window);
  /* Lancement de la boucle principale */
  gtk_main ();
  return EXIT_SUCCESS;
}
```

IV-D - Code source

chapitre4.zip

V - Afficher le contenue d'un fichier

V-A - Aperçu

Cliquez pour agrandir

V-B - Saisir du texte

Les choses sérieuses vont commencer, il s'agit de mettre en place le *widget* qui va nous permettre d'afficher et d'éditer du texte. Il s'agit de la classe *GtkTextView*. Grâce à **GTK+** la mise en place est toujours aussi simple :

main.c
```
GtkWidget *p_text_view = NULL;
/* ... */
/* Creation de la zone de texte */
p_text_view = gtk_text_view_new ();
gtk_box_pack_start (GTK_BOX (p_main_box), p_text_view, TRUE, TRUE, 0);
```

Comme il s'agit de la partie centrale de notre application, nous demandons à ce qu'elle prenne le plus de place possible (grâce à la fonction *gtk_box_pack_start*).

Puisque nous voulons afficher les boutons en dessous de la zone de texte, il faut ajouter ce morceau de code avant celui créant le bouton.

V-C - Ouvrir un fichier

Maintenant nous avons une belle zone de texte, il faut la remplir avec le contenu d'un fichier. Pour ce faire, nous allons commencer par ajouter un bouton ouvrir :

main.c
```
/* Creation du bouton "Ouvrir" */
{
  GtkWidget *p_button = NULL;

  p_button = gtk_button_new_from_stock (GTK_STOCK_OPEN);
  g_signal_connect (G_OBJECT (p_button), "clicked", G_CALLBACK (cb_open), p_text_view);
  gtk_box_pack_start (GTK_BOX (p_main_box), p_button, FALSE, FALSE, 0);
}
```

Si vous exécutez le programme maintenant (3) , vous remarquerez que les boutons sont ajoutés les uns en dessous des autres, ce qui diminue la zone de saisie (avec deux boutons ce n'est pas gênant mais au bout d'une dizaine ça risque d'être problématique). Pour pallier ce problème, nous allons utiliser un nouveau style de *GtkBox* : les *GtkButtonBox* qui sont prévus pour contenir des boutons (ça tombe plutôt bien :D). Donc pour éviter de diminuer la zone de saisie et comme nous avons de la place en largeur, nous allons utiliser un *GtkHButtonBox* qui va être inclus dans la *p_main_box*. Voici les modifications à effectuer :

main.c
```
GtkWidget *p_button_box = NULL;
/* ... */
/* Creation du conteneur pour les boutons */
p_button_box = gtk_hbutton_box_new ();
gtk_box_pack_start (GTK_BOX (p_main_box), p_button_box, FALSE, FALSE, 0);
/* ... */
```

main.c
```
    gtk_box_pack_start (GTK_BOX (p_button_box), p_button, FALSE, FALSE, 0);
    /* ... */
    gtk_box_pack_start (GTK_BOX (p_button_box), p_button, FALSE, FALSE, 0);
```

Maintenant il nous reste plus qu'à créer la fonction *cb_open* dans le fichier *callback.c*, pour l'instant nous nous contenterons d'ouvrir toujours le même fichier, nous verrons plus tard comment laisser le choix à l'utilisateur :

callback.c
```
#define DEFAULT_FILE "main.c"

void cb_open (GtkWidget *p_widget, gpointer user_data)
{
  open_file (DEFAULT_FILE, GTK_TEXT_VIEW (user_data));

  /* Parametre inutilise */
  (void)p_widget;
}
```

Vous aurez compris que l'on va déléguer tout le travail à la fonction *open_file*, qui devra ouvrir le fichier dont le nom est passé en premier paramètre pour l'afficher dans le *GktTextView*.

J'ai fait ce choix pour deux raisons : le code d'une fonction *callback* peut être très conséquent (surtout si l'on souhaite qu'elle affiche une boîte de dialogue), et aussi parce que *cb_open* n'est peut être pas la seule fonction à copier un fichier dans un *GtkTextView* (par exemple lors de la création d'un nouveau fichier, l'on peut vouloir copier un squelette de fichier).

Pour copier notre fichier, plutôt que d'utiliser la fonction *fgets*, nous allons nous servir de la **glib** (4) qui propose la fonction :

```
gboolean g_file_get_contents (const gchar *filename,  gchar **contents, gsize *length, GError
 **error);
```

Qui nous renvoit le contenu du fichier nommé *filename* dans le tampon *contents*. Il est possible de récupérer le nombre de caractères copiés et retourne TRUE en cas de succès, sinon FALSE et dans ce cas une structure de type *GError* est crée pour en savoir plus sur le type d'erreur rencontré :

callback.c
```
static void open_file (const gchar *file_name, GtkTextView *p_text_view)
{
  g_return_if_fail (file_name && p_text_view);
  {
    gchar *contents = NULL;

    if (g_file_get_contents (file_name, &contents, NULL, NULL))
    {
      /* Copie de contents dans le GtkTextView */
    }
    else
    {
     print_warning ("Impossible d'ouvrir le fichier %s\n", file_name);
    }
  }
}
```

Je pense qu'un certain nombre d'explications s'impose (surtout si je vais trop vite, n'hésitez pas à m'arrêter !) :

- *g_return_if_fail* est une macro qui peut être comparée à *assert* sauf qu'elle se contente de sortir de la fonction si l'expression passée en paramètre est fausse. C'est une méthode pratique pour vérifier la validité des paramètres (si la fonction doit retourne une valeur, utilisé plutôt *g_return_val_if_fail*, qui prend un second paramètre la valeur à retourner)
- Ensuite on essaie de récupérer le contenu de notre fichier
- Si cela échoue, nous le signalons à l'utilisateur à l'aide de la fonction *print_warning*.
- Le type *gchar* est une redéfinition du type *char* par la **glib** (il en va de même pour tous les autres types du C), privilégiez ces types lorsque vous utilisez des fonctions de cette bibliothèques.

Mais qu'est-ce donc cette fonction *print_warning* ? C'est une fonction que nous allons créer, semblable à *printf* qui signalera à l'utilisateur qu'une erreur non critique est survenue. Comme nous n'avons pas encore abordé les boîtes de dialogue pour afficher un message, il s'agira pour l'instant d'une simple redéfinition de *printf*. Pendant que nous sommes dans l'affichage des messages, nous allons aussi créer deux fonctions *print_info* et *print_error* qui vont respectivement afficher un message d'information et un message d'erreur critique (ce qui entraîne la terminaison du programme), tout ceci dans un nouveau fichier *error.c* :

error.h
```
#ifndef H_ERROR
#define H_ERROR

void print_info (char *, ...);
void print_warning (char *, ...);
void print_error (char *, ...);

#endif /* not H_ERROR */
```

error.c
```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include "error.h"

void print_info (char *format, ...)
{
  va_list va;

  va_start (va, format);
  printf ("Information : ");
  vprintf (format, va);
  printf ("\n");
}

void print_warning (char *format, ...)
{
  va_list va;

  va_start (va, format);
  fprintf (stderr, "Erreur : ");
  vfprintf (stderr, format, va);
  fprintf (stderr, "\n");
}

void print_error (char *format, ...)
{
  va_list va;

  va_start (va, format);
  fprintf (stderr, "Erreur fatale : ");
  vfprintf (stderr, format, va);
  fprintf (stderr, "\n");
  exit (EXIT_FAILURE);
```

error.c
```
    }
```

Pour en finir avec l'ouverture d'un fichier, il nous reste à copier *contents* dans le *GtkTextView*. En fait le *GtkTextView* ne gère que la forme, pour modifier le contenu, il faut passer par les *GtkTextBuffer*. Commençons donc par récupérer ce fameux *GtkTextBuffer* :

callback.c
```
        GtkTextBuffer *p_text_buffer = NULL;

        p_text_buffer = gtk_text_view_get_buffer (p_text_view);
```

Et pour finir on utilise la fonction :

```
    void gtk_text_buffer_insert (GtkTextBuffer *buffer, GtkTextIter *iter, const gchar *text, gint
     len);
```

Récapitulons : *buffer* est le *GtkTextBuffer* que l'on vient de récupérer, *text* c'est le texte à afficher et *len* la taille de ce dernier (ou -1 s'il s'agit d'une chaîne de caractères au sens du langage C).

Mais qu'est-ce donc ce *GtkTextIter* ? On peut voir cela comme un curseur virtuel qui indique la position à laquelle effectuer l'insertion de texte dans le *GtkTextBuffer* (5) . Allons voir ce que la documentation peut nous apprendre à leur sujet (6) :

```
    typedef struct {
      /* GtkTextIter is an opaque datatype; ignore all these fields.
       * Initialize the iter with gtk_text_buffer_get_iter_*
       * functions
       */
    } GtkTextIter;
```

Voilà, on a notre solution : suffit de rechercher les fonctions commençant par *gtk_text_buffer_get_iter_*, voici la liste :

```
    void gtk_text_buffer_get_iter_at_line_offset (GtkTextBuffer *buffer, GtkTextIter *iter, gint
     line_number, gint char_offset);
    void gtk_text_buffer_get_iter_at_offset (GtkTextBuffer *buffer, GtkTextIter *iter, gint
     char_offset);
    void gtk_text_buffer_get_iter_at_line (GtkTextBuffer *buffer, GtkTextIter *iter, gint line_number);
    void gtk_text_buffer_get_iter_at_line_index (GtkTextBuffer *buffer, GtkTextIter *iter, gint
     line_number, gint byte_index);
    void gtk_text_buffer_get_iter_at_mark (GtkTextBuffer *buffer, GtkTextIter *iter, GtkTextMark
     *mark);
    void gtk_text_buffer_get_iter_at_child_anchor (GtkTextBuffer *buffer, GtkTextIter *iter,
     GtkTextChildAnchor *anchor);
```

La fonction *gtk_text_buffer_get_iter_at_line* fait parfaitement l'affaire :

callback.c
```
        GtkTextIter iter;

        /* ... */
        gtk_text_buffer_get_iter_at_line (p_text_buffer, &iter, 0);
        gtk_text_buffer_insert (p_text_buffer, &iter, contents, -1);
```

Cependant si vous ne voulez pas avoir de problèmes avec le codage des caractères, il vaut mieux convertir le codage local vers utf8. Voici le résultat final :

```
        gchar *utf8 = NULL;
        GtkTextIter iter;
        GtkTextBuffer *p_text_buffer = NULL;

        p_text_buffer = gtk_text_view_get_buffer (p_text_view);
        gtk_text_buffer_get_iter_at_line (p_text_buffer, &iter, 0);
        utf8 = g_locale_to_utf8 (contents, -1, NULL, NULL, NULL);
        g_free (contents), contents = NULL;
        gtk_text_buffer_insert (p_text_buffer, &iter, utf8, -1);
        g_free (utf8), utf8 = NULL;
```

V-D - La petite touche finale

Pour finir cette laborieuse partie sur une petite touche esthétique, nous allons demander à **GTK+** de nous lancer l'application en plein écran. Pour se faire, il suffit d'ajouter, lors de la création de la fenêtre principale :

main.c
```
    gtk_window_maximize (GTK_WINDOW (p_window));
```

On peut même se permettre une pointe d'excentricité en modifiant le titre de notre fenêtre :

main.c
```
    gtk_window_set_title (GTK_WINDOW (p_window), "Editeur de texte en GTK+");
```

V-E - Code source

chapitre5.zip

VI - Choisir un fichier

VI-A - Aperçu

Cliquez pour agrandir

VI-B - Utilisation d'un GtkFileChooserFile

Jusqu'à présent l'utilisateur n'avait pas le choix quant au fichier à ouvrir, heureusement **GTK+** vient à notre aide grâce au *widget GtkFileChooserFile* qui est tout simplement une boîte de dialogue qui propose à l'utilisateur de sélectionner un fichier dans son arborescence disque.

Commençons par créer notre boîte de dialogue dans la fonction *cb_open* :

callback.c
```
void cb_open (GtkWidget *p_widget, gpointer user_data)
{
  GtkWidget *p_dialog = NULL;

  p_dialog = gtk_file_chooser_dialog_new ("Ouvrir un fichier", NULL,
                                          GTK_FILE_CHOOSER_ACTION_OPEN,
                                          GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                          GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT,
                                          NULL);
  /* ... */
}
```

Cette fonction nécessite le titre de la boîte de dialogue, une fenêtre parente, le type d'action (ici, on souhaite ouvrir un fichier, **GTK+** autorisera l'utilisateur à sélectionner uniquement les fichiers existants). Pour finir, il s'agit d'un couple de valeurs GTK_STOCK_ITEM/GTK_STOCK_RESPONSE qui permet de créer un bouton utilisant le stock ID spécifié et, lorsque celui-ci est cliqué, la fonction servant à lancer la boîte de dialogue retournera le réponse ID correspondant. Il existe une série de valeurs définies par **GTK+**, qu'il est conseillé d'utiliser :

```
typedef enum
{
  /* GTK returns this if a response widget has no response_id,
   * or if the dialog gets programmatically hidden or destroyed.
   */
  GTK_RESPONSE_NONE = -1,

  /* GTK won't return these unless you pass them in
   * as the response for an action widget. They are
   * for your convenience.
   */
  GTK_RESPONSE_REJECT = -2,
  GTK_RESPONSE_ACCEPT = -3,

  /* If the dialog is deleted. */
  GTK_RESPONSE_DELETE_EVENT = -4,

  /* These are returned from GTK dialogs, and you can also use them
   * yourself if you like.
   */
  GTK_RESPONSE_OK     = -5,
  GTK_RESPONSE_CANCEL = -6,
  GTK_RESPONSE_CLOSE  = -7,
  GTK_RESPONSE_YES    = -8,
```

```
    GTK_RESPONSE_NO      = -9,
    GTK_RESPONSE_APPLY   = -10,
    GTK_RESPONSE_HELP    = -11
} GtkResponseType;
```

Nous indiquons la fin des couples stock id/réponse id avec la valeur NULL.

Une fois la boîte de dialogue créee, on demande à **GTK+** de l'afficher grâce à la fonction *gtk_dialog_run*, puis on récupère sa valeur de retour, qui correspond au bouton cliqué par l'utilisateur :

callback.c
```
if (gtk_dialog_run (GTK_DIALOG (p_dialog)) == GTK_RESPONSE_ACCEPT)
{
  /* ... */
}
```

Ici ce qui nous intéresse, c'est lorsque que l'utilisateur clique sur le bouton ouvrir (donc *gtk_dialog_run* renvoie *GTK_RESPONSE_ACCEPT*), dans ce cas, il nous suffit de récupérer le nom du fichier sélectionné puis de l'ouvrir :

callback.c
```
gchar *file_name = NULL;

file_name = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (p_dialog));
open_file (file_name, GTK_TEXT_VIEW (user_data));
g_free (file_name), file_name = NULL;
```

Pour finir, dans tous les cas, on n'oublie pas de détruire la boîte de dialogue :

callback.c
```
gtk_widget_destroy (p_dialog);
```

Et voilà le travail : l'utilisateur peut ouvrir le fichier qu'il souhaite.

VI-C - Code source

chapitre6.zip

VII - Interlude

Avant d'aller plus loin dans l'amélioration de l'interface de notre éditeur, il est nécessaire de modifier son fonctionnement interne (ne vous inquiétez je n'ai pas dit chambouler), en effet, souvenez-vous dans l'introduction j'ai évoqué la possibilités d'ouvrir plusieurs documents grâce à un système d'onglets. Pour cela, il faut sauvegarder les propriétés de chaque document ouvert. Pris à temps ce genre de modification n'est pas trop lourde, mais si nous attendons, cela risque de devenir un vrai casse tête !

Pour cela nous allons utiliser une structure de donnée pour chaque document ouvert qui devra garder en mémoire le chemin complet du fichier (ou NULL si le document n'est pas encore sauvegarder), s'il à était modifier depuis la dernière sauvegarde et le *GtkTextView* qui sert à son affichage. Voici donc la structure qui va nous servir :

document.h
```
typedef struct
{
  gchar *chemin;
  gboolean sauve;
  GtkTextView *p_text_view;
} document_t;
```

Sachant que nous serons amené à stocker plusieurs occurrences de cette structure, il serait bon de réfléchir dès maintenant à la structure de donnée à utiliser pour les stocker. La première idée qui vient à l'esprit est un tableau alloué dynamiquement, cependant l'utilisateur peut souhaiter fermer un document qui se trouve au milieu, on est alors obligé de décaler toutes les cellules en amont. Dans ce cas il parait naturel d'utiliser une liste chaînée. Par chance la **glib** propose une bibliothèque de gestion des listes chaînées, cela nous fera gagner du temps le moment venu. Pour l'instant, on se contente de créer une structure de données qui contiendra tous les documents ouverts (stockée dans une GList) et un pointeur sur le document actif (actuellement comme nous n'avons qu'un document ouvert en même temps, on manipulera uniquement ce pointeur) :

document.h
```
typedef struct
{
  GList *tous;
  document_t *actif;
} docs_t;
```

Maintenant se pose le problème de savoir comment transmettre cette structure. On pourrait se servir du paramètre *user_data* présent dans toutes les fonctions *callback* mais certaines fonctions utilise déjà ce paramètre (la fonction *cb_open* par exemple), il faudrait créer un champs supplémentaire dans notre structure *documents_s*. Une solution plus simple est de créer une variable globale à l'ensemble du programme. Ceux qui passe régulièrement sur le forum C savent que c'est fortement déconseillé, mais ici nous nous trouvons dans l'une des rares exceptions à la règle. De toute façon cela revient au même que de passer l'adresse de notre structure à toutes les fonctions *callback*. Nous définissons donc un nouveau fichier d'en-tête :

document.h
```
#ifndef H_DOCUMENT
#define H_DOCUMENT

#include <gtk/gtk.h>

typedef struct
{
  gchar *chemin;
  gboolean sauve;
  GtkTextView *p_text_view;
} document_t;
```

document.h
```
typedef struct
{
  GList *tous;
  document_t *actif;
} docs_t;

extern docs_t docs;

#endif /* not H_DOCUMENT */
```

Et dans le fichier *main.c* :

main.c
```
#include "document.h"

docs_t docs = {NULL, NULL};
```

Pour l'instant la seule fonction qui modifie l'état d'un document est la fonction *open_file*, il faut donc inclure *document.h* dans *callback.c* et modifier légèrement la fonction pour enregistrer le document ouvert :

callback.c
```
      if (g_file_get_contents (file_name, &contents, NULL, NULL))
      {
        /* Pour l'instant il faut allouer la memoire, par la suite on modifira
           simplement le pointeur */
        docs.actif = g_malloc (sizeof (*docs.actif));
        docs.actif->chemin = g_strdup (file_name);
        /* Pour l'instant, on se contente de stocker le seul GtkTextView qui existe,
          par la suite, il faudra en creer un nouveau ! */
        docs.actif->p_text_view = p_text_view;
        /* Le document vient d'etre ouvert, il n'est donc pas modifie */
        docs.actif->sauve = TRUE;
        /* ... */
      }
```

Ne soyez pas choqué si je ne teste pas le retour de la fonction g_malloc pour vérifier qu'il n'est pas NULL. En effet, cette dernière met fin à l'application si l'allocation échoue.

VII-A - Code source

chapitre7.zip

VIII - Sauvegarder les modification

VIII-A - Aperçu

Cliquez pour agrandir

VIII-B - Enregistrer

Avant de pouvoir sauvegarder un document, il faut qu'il soit modifié. Comment savoir que le contenu du fichier a été modifier ? Grâce au signal *changed* du *GtkTextBuffer* que nous interceptons grâce à la fonction *cb_modifie* :

main.c
```
    {
      GtkTextBuffer *p_text_buffer = NULL;

      p_text_buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (p_text_view));
      g_signal_connect (G_OBJECT (p_text_buffer), "changed", G_CALLBACK (cb_modifie), NULL);
    }
```

Cette dernière modifie simplement le champ *sauve* du document :

callback.c
```
  void cb_modifie (GtkWidget *p_widget, gpointer user_data)
  {
    if (docs.actif)
    {
      docs.actif->sauve = FALSE;
    }
  }
```

Pour la sauvegarde, il faut distinguer deux cas : soit il s'agit de la première sauvegarde du fichier, il faut alors demander le nom du fichier à l'utilisateur (cela ressemble fortement à l'ouverture d'un fichier), soit le fichier est déjà présent sur le disque, il suffit de remplacer son contenu par celui du *GtkTextViewer*. Nous avons convenu qu'un fichier qui n'était pas encore enregistré avait sa propriété *chemin* à NULL.

Bien sûr cela n'a lieu que si un fichier est ouvert et qu'il a été modifié depuis sa dernière sauvegarde :

callback.c
```
  void cb_save (GtkWidget *p_widget, gpointer user_data)
  {
    if (docs.actif)
    {
      if (!docs.actif->sauve)
      {
        /* Le fichier n'a pas encore ete enregistre */
        if (!docs.actif->chemin)
        {
          GtkWidget *p_dialog = NULL;

          p_dialog = gtk_file_chooser_dialog_new ("Sauvegarder le fichier", NULL,
                                                  GTK_FILE_CHOOSER_ACTION_SAVE,
                                                  GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                                  GTK_STOCK_SAVE, GTK_RESPONSE_ACCEPT,
                                                  NULL);
          if (gtk_dialog_run (GTK_DIALOG (p_dialog)) == GTK_RESPONSE_ACCEPT)
          {
```

callback.c
```
            docs.actif->chemin = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (p_dialog));
          }
          gtk_widget_destroy (p_dialog);
        }
        /* Soit le fichier a deja ete enregistre, soit l'utilisateur vient de nous
           fournir son nouvel enplacement, on peut donc l'enregistrer */
        if (docs.actif->chemin)
        {
          /* ... */
        }
      }
    }
    else
    {
      print_warning ("Aucun document ouvert");
    }
  }
```

Il nous reste plus qu'à récupérer le contenu du *GtkTextViewer* et le copier dans le fichier *chemin* sans oublier de reconvertir le texte dans le codage local :

callback.c
```
          FILE *fichier = NULL;

          fichier = fopen (docs.actif->chemin, "w");
          if (fichier)
          {
            gchar *contents = NULL;
            gchar *locale = NULL;
            GtkTextIter start;
            GtkTextIter end;
            GtkTextBuffer *p_text_buffer = NULL;

            p_text_buffer = gtk_text_view_get_buffer (docs.actif->p_text_view);
            gtk_text_buffer_get_bounds (p_text_buffer, &start, &end);
            contents = gtk_text_buffer_get_text (p_text_buffer, &start, &end, FALSE);
            locale = g_locale_from_utf8 (contents, -1, NULL, NULL, NULL);
            g_free (contents), contents = NULL;
            fprintf (fichier, "%s", locale);
            g_free (locale), locale = NULL;
            fclose (fichier), fichier = NULL;
            docs.actif->sauve = TRUE;
          }
          else
          {
            print_warning ("Impossible de sauvegarder le fichier %s", docs.actif->chemin);
          }
```

Le dernier paramètre de la fonction *gtk_text_buffer_get_text* est mis à *FALSE* car nous ne voulons pas enregistrer les caractères invisibles présent dans le *GtkTextBuffer*. Ces caractères servent à mettre en forme le texte (taille, couleur...), nous pourrions créer un nouveau format de fichier pour enregistrer la mise en forme.

VIII-C - Enregistrer sous

Pour enregistrer notre document sous un autre nom, il suffit de faire croire à *cb_save* que le document n'a pas encore était enregistré, tout simplement en changeant *chemin* à NULL et *sauve* à FALSE :

callback.c
```
  void cb_saveas (GtkWidget *p_widget, gpointer user_data)
  {
```

callback.c
```
    if (docs.actif)
    {
      document_t tmp = *docs.actif;

      docs.actif->chemin = NULL;
      docs.actif->sauve = FALSE;
      cb_save (p_widget, user_data);
      if (!docs.actif->sauve)
      {
        (*docs.actif) = tmp;
      }
    }
    else
    {
      print_warning ("Aucun document ouvert");
    }
  }
```

Il ne faut pas oublier que l'utilisateur peut annuler l'enregistrement, de ce fait, nous sauvegardons l'état du document et si la sauvegarde n'a pas eu lieu, on le restaure.

VIII-D - Code source

chapitre8.zip

IX - Créer un nouveau document

IX-A - Aperçu

Cliquez pour agrandir

IX-B - Nouveau fichier

Maintenant que l'utilisateur peut ouvrir et fermer un fichier, il est intéressant de lui donner la possibilité d'en créer un nouveau. Je ne reviens pas sur la création du bouton (ça devrait déjà être fait :D), passons directement à la fonction *cb_new* :

callback.c
```
void cb_new (GtkWidget *p_widget, gpointer user_data)
{
  /* Pour l'instant il faut allouer la memoire, par la suite on modifiera
     simplement le pointeur */
  docs.actif = g_malloc (sizeof (*docs.actif));
  docs.actif->chemin = NULL;
  /* Pour l'instant, on se contente de stocker le seul GtkTextView qui existe,
    par la suite, il faudra en creer un nouveau ! */
  docs.actif->p_text_view = GTK_TEXT_VIEW (user_data);
  /* Le document vient d'etre creer, il n'est donc pas modifie */
  docs.actif->sauve = TRUE;
  gtk_widget_set_sensitive (GTK_WIDGET (docs.actif->p_text_view), TRUE);
}
```

Je ne sais pas vous mais pour ma part j'ai fait un copier/coller de la fonction *open_file* ! Et oui, ouvrir un document peut se résumer à créer un document vide puis remplir le *GtkTextView* avec le fichier souhaité ! On peut donc simplifier notre fonction *open_file* ainsi :

callback.c
```
static void open_file (const gchar *file_name, GtkTextView *p_text_view)
{
  g_return_if_fail (file_name && p_text_view);
  {
    gchar *contents = NULL;

    if (g_file_get_contents (file_name, &contents, NULL, NULL))
    {
      /* Copie de contents dans le GtkTextView */
      GtkTextIter iter;
      GtkTextBuffer *p_text_buffer = NULL;

      cb_new (NULL, p_text_view);
      gtk_widget_set_sensitive (GTK_WIDGET (docs.actif->p_text_view), TRUE);
      p_text_buffer = gtk_text_view_get_buffer (p_text_view);
      gtk_text_buffer_get_iter_at_line (p_text_buffer, &iter, 0);
      gtk_text_buffer_insert (p_text_buffer, &iter, contents, -1);
      /* Nous sommes obliges de remetre sauve a TRUE car l'insertion du contenu
         du fichier dans le GtkTextView a appele cb_modfie */
      docs.actif->sauve = TRUE;
    }
    else
    {
     print_warning ("Impossible d'ouvrir le fichier %s\n", file_name);
    }
  }
```

callback.c
```
    }
```

C'est beau la factorisation du code ! Par contre nous sommes obligés de remettre *sauve* à TRUE car nous avons modifié le *GtkTextBuffer*.

IX-C - Code source

chapitre9.zip

X - Fermer

X-A - Aperçu

Cliquez pour agrandir

X-B - Fermer un fichier

Maintenant que nous allouons de la mémoire, il faut la libérer à un endroit. Logiquement cela va être fait à la fermeture du document, en guise d'exercice, je vous laisse créer le bouton dans notre boîte à bouton...

...

Allez je vous aide, le stock id correspondant est GTK_STOCK_CLOSE.

Voilà maintenant, si tout c'est bien passé, vous devriez être arrivé dans le fichier *callback.c* avec quelque chose ressemblant à :

callback.c
```
void cb_close (GtkWidget *p_widget, gpointer user_data)
{
  /* ... */
}
```

Lorsque l'on ferme un document, il faut vider le *GtkTextView* (avec les onglets, on se contentera de le supprimer), pour cela, il faut récupérer le début et la fin du *GtkTextBuffer* et demander à **GTK+** de supprimer tout ce qui ce trouve entre les deux itérateurs :

callback.c
```
/* Avant de fermer, il faut verifier qu'un document a bien ete ouvert */
if (docs.actif)
{
  GtkTextIter start;
  GtkTextIter end;
  GtkTextBuffer *p_text_buffer = NULL;

  p_text_buffer = gtk_text_view_get_buffer (docs.actif->p_text_view);
  gtk_text_buffer_get_bounds (p_text_buffer, &start, &end);
  gtk_text_buffer_delete (p_text_buffer, &start, &end);
  /* ... */
}
else
{
  print_warning ("Aucun document ouvert");
}
```

Bien sûr il ne faut pas oublier de libérer la mémoire

callback.c
```
  g_free (docs.actif->chemin), docs.actif->chemin = NULL;
  docs.actif->p_text_view = NULL;
  g_free (docs.actif), docs.actif = NULL;
```

Pour symboliser la fermeture d'un document, nous désactivons le *GtkTextView* lors de la fermeture (ne pas oublier de le faire aussi dans *main.c*) :

callback.c
```
       gtk_widget_set_sensitive (GTK_WIDGET (docs.actif->p_text_view), FALSE);
```

Et nous le réactivons lors de l'ouverture si celle si réussie :

callback.c
```
       gtk_widget_set_sensitive (GTK_WIDGET (docs.actif->p_text_view), TRUE);
```

Lorsque l'utilisateur quitte l'application, il faut bien sûr fermer le document, s'il y en a un ouvert :

```
void cb_quit (GtkWidget *p_widget, gpointer user_data)
{
  if (docs.actif)
  {
    cb_close (p_widget, user_data);
  }
  gtk_main_quit();
}
```

Et aussi lorsqu'il créé un nouveau document (et par conséquent lorsqu'il ouvre un fichier puisque l'on fait appel à *cb_new*) :

```
void cb_new (GtkWidget *p_widget, gpointer user_data)
{
  if (docs.actif)
  {
    cb_close (p_widget, user_data);
  }
  /* ... */
}
```

X-C - Enregistrer avant de fermer

Si le document a été modifié depuis la dernière sauvegarde, généralement le programme le signale à l'utilisateur et lui propose de sauvegarder ou d'annuler la fermeture. Pour se faire, nous devons modifier la fonction *cb_close*, avant de fermer le document, nous allons afficher une boîte de dialogue.

Pour créer une boîte de dialogue simplement, il existe la classe *GtkDialog* et comme nous avons besoin de boutons (pour que l'utilisateur fasse son choix), nous allons créer notre boîte avec la fonction :

```
GtkWidget *gtk_dialog_new_with_buttons (const gchar *title, GtkWindow *parent, GtkDialogFlags
 flags, const gchar *first_button_text, ...);
```

Nous retrouvons les mêmes options que pour le *GtkFileChooserDialog*, mis à part option du type *GtkDialogFlags* :

```
typedef enum
{
  GTK_DIALOG_MODAL               = 1 << 0, /* appel gtk_window_set_modal (win, TRUE) */
  GTK_DIALOG_DESTROY_WITH_PARENT = 1 << 1, /* appel gtk_window_set_destroy_with_parent () */
  GTK_DIALOG_NO_SEPARATOR        = 1 << 2  /* Pas de barre de separation au dessus des boutons */
} GtkDialogFlags;
```

Nous nous contenterons de rendre notre fenêtre modale (7) .

Pour avoir accès à notre fenêtre principale, nous ajoutons un champs *p_main_window* à notre structure globale *docs_t* que nous initialisons dans la fonction *main* :

main.c
```
    docs.p_main_window = GTK_WINDOW (p_window);
```

Il nous reste plus qu'a créer notre boîte de dialogue avec trois boutons : Oui, Non, Annuler :

callback.c
```
      if (!docs.actif->sauve)
      {
        GtkWidget *p_dialog = NULL;

        p_dialog = gtk_dialog_new_with_buttons ("Sauvegarder",
                                                docs.p_main_window,
                                                GTK_DIALOG_MODAL,
                                                GTK_STOCK_YES, GTK_RESPONSE_YES,
                                                GTK_STOCK_NO, GTK_RESPONSE_NO,
                                                GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL, NULL);
        /* ... */
```

Nous obtenons donc une structure de type *GtkDialog* :

```
  typedef struct {
    GtkWidget *vbox;
    GtkWidget *action_area;
  } GtkDialog;
```

Nous remarquons que cette structure contient deux membres qui permettent d'accéder à une *GtkBox*, où nous allons ajouter le contenu de la fenêtre, et à la partie contenant les boutons.

Ensuite la construction de la fenêtre est identique à celle de la fenêtre principale, ici nous nous contenterons d'ajouter un *GtkLabel* :

callback.c
```
        GtkWidget *p_label = NULL;
        /* ... */
        p_label = gtk_label_new ("Voulez-vous sauvegarder les modifications ?");
        gtk_box_pack_start (GTK_BOX (GTK_DIALOG (p_dialog)->vbox), p_label, TRUE, TRUE, 0);
```

Une fois notre fenêtre prête, il suffit de faire appel à la fonction *gtk_dialog_run* qui va afficher notre *GtkDialog* et nous retourner le réponse id correspondant au bouton cliqué par l'utilisateur :

callback.c
```
        switch (gtk_dialog_run (GTK_DIALOG (p_dialog)))
        {
          case GTK_RESPONSE_YES:
            cb_save (p_widget, user_data);
          break;
          case GTK_RESPONSE_NO:
          break;
          case GTK_RESPONSE_CANCEL:
            gtk_widget_destroy (p_dialog);
            return;
          break;
        }
```

callback.c
```
          gtk_widget_destroy (p_dialog);
        }
        /* ... */
```

Si l'utilisateur clique sur non, on ne fait rien (le document sera simplement fermé), sur oui, on enregistre avant de fermer et pour finir, s'il annule on quitte la fonction.

X-D - Code source

chapitre10.zip

XI - Les barres de défilement

XI-A - Aperçu

Cliquez pour agrandir

XI-B - Ajouter des barres de défilement

Après le dernier chapitre, quelque peu laborieux, voici une partie plus simple mais qui va rendre notre application plus pratique. En effet, si vous avez essayé d'ouvrir un fichier de grande taille, vous avez pu remarquer que pour pouvoir lire la fin du fichier, il fallait utiliser les touches du clavier : pas très convivial.

Pour rendre la navigation plus aisée, on utilise des barres de défilement :

main.c
```
	GtkWidget *p_scrolled_window = NULL;

	p_scrolled_window = gtk_scrolled_window_new (NULL, NULL);
	gtk_box_pack_start (GTK_BOX (p_main_box), p_scrolled_window, TRUE, TRUE, 0);

	/* Creation de la zone de texte */
	/* ... */
	gtk_container_add (GTK_CONTAINER (p_scrolled_window), p_text_view);
```

Le constructeur de notre *GtkScrolledWindow* prend en argument deux *GtkAdjustment* qui permettent de définir différentes propriétés de la barre de défilement (taille d'une page, la position de départ...), nous laissons **GTK+** faire en passant **NULL**.

C'est un bon début mais esthétiquement on peut faire mieux : même s'il n'est pas utile d'avoir une barre de défilement, elle est quand même affichée. On peut demander à **GTK+** de les afficher que si cela est nécessaire :

main.c
```
	gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (p_scrolled_window), GTK_POLICY_AUTOMATIC,
    GTK_POLICY_AUTOMATIC);
```

En fait nous avons de la chance car la classe *GtkTextView* fait partie des *widget* qui supporte de façon native les barres de défilement. Comment le savoir ? Ce genre de *widget* possède une ou deux propriétés de type *GtkAdjustment* (une pour la barre verticale, l'autre pour la barre horizontale). Actuellement seulement trois classes en sont capables : les *GtkTextView*, les *GtkTreeView* et les *GtkLayout*.

Comment faire pour les autres *widgets* ? Il faut passer par une classe adaptateur *GtkViewport* afin de créer les *GtkAdjustment*.

XI-C - Code source

chapitre11.zip

XII - Les menus

XII-A - Aperçu

Cliquez pour agrandir

XII-B - Création du menu

Pour simplifier nous avons utilisé des boutons pour permettre à l'utilisateur de créer un document, ouvrir un fichier... Mais il est plus courant d'utiliser un menu pour afficher ces options. **GTK+** propose trois manières de créer un menu :

- *GtkItemFactory* : cette méthode est obsolète depuis la version 2.4 de **GTK+**
- *GtkUIManager* : c'est ce qu'on appel une usine à gaz, pour en savoir plus, vous pouvez vous reporter au tutoriel **Utilisation de GtkUIManager**
- *GtkMenu* : c'est ce *widget* que nous allons étudier, il permet de créer un menu manuellement (à l'opposé de *GtkUIManager*).

Un menu, selon **GTK+**, est composé de plusieurs éléments :

- *GtkMenuBar* : la barre de menu elle-même
- *GtkMenu* : la partie déroulante qui contient les différents éléments
- *GtkMenuItem* : c'est sur ce *widget* que l'utilisateur clique pour lancer une action.

Pour commencer, faisons l'inventaire de ce que nous avons besoin :

- Un *GtkMenuBar* qui sert de base au menu
- Un *GtkMenu*, pour commencer nous nous aurons d'un menu *Fichier*
- Six *GtkMenuItem* pour remplacer nos six boutons.

Commençons par la création du menu, nous mettons ce code dans un nouveau fichier dont seul la fonction *menu_new* sera accessible :

menu.c
```
GtkMenuBar *menu_new (gpointer user_data)
{
  GtkWidget *p_menu_bar = NULL;

  p_menu_bar = gtk_menu_bar_new ();
  /* ... */
  return GTK_MENU_BAR (p_menu_bar);
}
```

Le paramètre *user_data* nous servira lors de la connexion des callbacks (nous en avons besoin pour les fonctions *cb_new* et *cb_open*).

Ensuite, nous allons créer notre sous menu *Fichier* :

menu.c
```
  /* Menu "Fichier" */
```

menu.c
```
    {
      GtkWidget *p_menu = NULL;
      GtkWidget *p_menu_item = NULL;

      p_menu = gtk_menu_new ();
      p_menu_item = gtk_menu_item_new_with_mnemonic ("_Fichier");
      /* ... */
      gtk_menu_item_set_submenu (GTK_MENU_ITEM (p_menu_item), p_menu);
      gtk_menu_shell_append (GTK_MENU_SHELL (p_menu_bar), p_menu_item);
    }
    return GTK_MENU_BAR (p_menu_bar);
```

Un sous menu est en fait un *GtkMenuItem* auquel on assigne un *GtkMenu*. Il faut bien sûr terminer la création du sous-menu en l'ajoutant à la barre de menu.

Pour finir, nous créons les éléments du menu, les *GtkItemMenu*. Il existe plusieurs types d'éléments (comparable aux différents types de bouton), pour commencer nous n'utiliserons que le type de base. Comme cette opération est répétitive, nous allons créer une fonction pour le faire :

menu.c
```
  static void menu_item_new (GtkMenu *p_menu, const gchar *title, GCallback callback, gpointer
   user_data)
  {
    GtkWidget *p_menu_item = NULL;

    p_menu_item = gtk_menu_item_new_with_mnemonic (title);
    gtk_menu_shell_append (GTK_MENU_SHELL (p_menu), p_menu_item);
    g_signal_connect (G_OBJECT (p_menu_item), "activate", callback, user_data);
  }
```

Pour permettre à l'utilisateur d'accéder aux différents éléments du menu à l'aide de la touche <Alt>, nous utilisons des mnémoniques, par exemple pour quitter l'éditeur il suffit de faite Alt+F puis Q. Et voici le code pour créer nos six éléments du menu :

menu.c
```
    menu_item_new (GTK_MENU (p_menu), "_Nouveau", G_CALLBACK (cb_new), user_data);
    menu_item_new (GTK_MENU (p_menu), "_Ouvrir", G_CALLBACK (cb_open), user_data);
    menu_item_new (GTK_MENU (p_menu), "_Enregistrer", G_CALLBACK (cb_save), user_data);
    menu_item_new (GTK_MENU (p_menu), "Enregistrer _sous", G_CALLBACK (cb_saveas), user_data);
    menu_item_new (GTK_MENU (p_menu), "_Fermer", G_CALLBACK (cb_close), user_data);
    menu_item_new (GTK_MENU (p_menu), "_Quitter", G_CALLBACK (cb_quit), user_data);
```

Voilà, notre menu est créé ! il nous reste plus qu'à l'intégrer à notre fenêtre :

main.c
```
    gtk_box_pack_start (GTK_BOX (p_main_box), GTK_WIDGET (menu_new (p_text_view)), FALSE, FALSE, 0);
```

Le problème c'est que nous avons besoin de passer le *GtkTextView* lors de la création du menu (qui est utilisé par les fonctions *cb_new* et *cb_open*) or celui-ci est créé après le menu (puisque nous créons les *widgets* dans l'ordre où il faut les intégrer à l'interface (8)), nous devons créer le *GtkTextView* (seul l'appel à *gtk_text_view_new* est déplacé).

XII-C - Code source

chapitre12.zip

XIII - Les barres d'outils

XIII-A - Aperçu

Cliquez pour agrandir

XIII-B - Création d'une barre d'outils

La création d'une barre d'outils ressemble fort à celle d'un menu, en plus simple puisqu'il n'y a pas de sous menu : on crée notre barre d'outils (*gtk_toolbar_new*) puis les éléments de la barre, pour cela on utilise des boutons (*gtk_tool_button_new_from_stock*), que l'on insére dans la barre (*gtk_toolbar_insert*). Pas conséquent notre fichier *barreoutils.c* ressemble à *menu.c* :

barreoutils.c
```
#include <gtk/gtk.h>
#include "callback.h"
#include "barreoutils.h"

static void toolbar_item_new (GtkToolbar *, const gchar *, GCallback, gpointer);

GtkToolbar *toolbar_new (gpointer user_data)
{
  GtkWidget *p_toolbar = NULL;

  p_toolbar = gtk_toolbar_new ();
  /* ... */
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_NEW, G_CALLBACK (cb_new), user_data);
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_OPEN, G_CALLBACK (cb_open), user_data);
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_SAVE, G_CALLBACK (cb_save), user_data);
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_SAVE_AS, G_CALLBACK (cb_saveas), user_data);
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_CLOSE, G_CALLBACK (cb_close), user_data);
  toolbar_item_new (GTK_TOOLBAR (p_toolbar), GTK_STOCK_QUIT, G_CALLBACK (cb_quit), user_data);
  return GTK_TOOLBAR (p_toolbar);
}

static void toolbar_item_new (GtkToolbar *p_toolbar, const gchar *stock_id, GCallback callback,
 gpointer user_data)
{
  GtkToolItem *p_tool_item = NULL;

  p_tool_item = gtk_tool_button_new_from_stock (stock_id);
  g_signal_connect (G_OBJECT (p_tool_item), "clicked", callback, user_data);
  gtk_toolbar_insert (p_toolbar, p_tool_item, -1);
}
```

Le code n'est pas complet car lorsque j'exécute le programme, **GTK+** affiche, en plus des icônes, le texte sous les boutons. Pour modifier cela, il suffit d'ajouter une ligne de code :

barreoutils.c
```
    gtk_toolbar_set_style (GTK_TOOLBAR (p_toolbar), GTK_TOOLBAR_ICONS);
```

*Pourquoi gtk_tool_button_new_from_stock retourne un GtkToolItem et non un GtkWidget comme nous avons été habitué jusqu'à présent. Il s'agit d'une bizarrerie de **GTK+** dont je ne connais pas la raison.*

*Pour anticiper un changement d'interface (dans **GTK+** 3.0 peut être ?), nous aurions pu écrire :*

```
Gtkwidget *p_tool_item = NULL;

p_tool_item = GTK_WIDGET (gtk_tool_button_new_from_stock (stock_id));
g_signal_connect (G_OBJECT (p_tool_item), "clicked", callback, user_data);
gtk_toolbar_insert (p_toolbar, GTK_TOOL_ITEM (p_tool_item), -1);
```

XIII-C - Code source

chapitre13.zip

XIV - Les racourcis clavier

XIV-A - Aperçu

Cliquez pour agrandir

XIV-B - Mise en place des raccourcis clavier

Il ne faut pas confondre les raccourcis clavier et les mnémoniques, ces derniers permettent d'accéder à un élément séquentiellement alors que les raccourcis appellent une fonction si l'utilisateur appuie simultanément sur une ou plusieurs touches (généralement de la forme <Modificateur>Touche, où Modificateur représente les touches Shift, Alt et Control). Ce raccourci peut être attaché à un élément du menu mais ceci n'est pas obligatoire.

Les raccourcis sont regroupés en groupe dans un *GtkAccelGroup* qu'il faut commencer par créer :

raccourcis.c
```
#include <gtk/gtk.h>
#include "callback.h"
#include "raccourcis.h"

GtkAccelGroup *accel_group_new (gpointer user_data)
{
  GtkAccelGroup *p_accel_group = NULL;

  p_accel_group = gtk_accel_group_new ();
  /* ... */
  return p_accel_group;
}
```

Vous commencez à avoir l'habitude de cette organisation, nous allons donc créer une fonction qui va se charger d'ajouter un raccourci au groupe :

raccourcis.c
```
static void accelerator_new (GtkAccelGroup *p_accel_group, const gchar *accelerator, const gchar
 *accel_path,
                             GCallback callback, gpointer user_data)
{
  guint key;
  GdkModifierType mods;
  GClosure *closure = NULL;

  gtk_accelerator_parse (accelerator, &key, &mods);
  closure = g_cclosure_new (callback, user_data, NULL);
  gtk_accel_group_connect (p_accel_group, key, mods, GTK_ACCEL_VISIBLE, closure);
  gtk_accel_map_add_entry (accel_path, key, mods);
}
```

La fonction *gtk_accelerator_parse* permet de décomposer une chaîne de caractères de la forme "<Control>F1" ou encore "<Alt>A" en numéro de touche et modificateur.

En plus des touches, pour créer un raccourci clavier, nous avons besoin d'une fonction à appeler lorsque l'utilisateur appuie sur les touches spécifiées. *gtk_accel_group_connect* attend une fonction du type *GClosure*, regardons la documentation de **gobject** pour savoir à quoi cela correspond :

```
typedef struct {
```

```
} GClosure;
```

Bon pas très instructif :(Heureusement en regardant le constructeur de la classe *GClosure* on retombe sur des choses connues :

```
GClosure *g_cclosure_new (GCallback callback_func, gpointer user_data, GClosureNotify
 destroy_data);
```

Le dernier paramètre est une fonction qui sera appelée pour détruire l'objet *user_data* lorsqu'il ne sera plus utilisé.

Voilà nous pouvons déjà connecter le raccourci clavier à notre fonction *callback*.

Pour finir, pour associer un élément du menu à un raccourci clavier, nous avons besoin de l'ajouter à la carte des raccourcis, cette carte est unique et spécifique à chaque application :

```
void gtk_accel_map_add_entry (const gchar *accel_path, guint accel_key, GdkModifierType
 accel_mods);
```

Il nous manque juste le paramètre *accel_path*. Il s'agit, comme son nom le laisse penser, d'un chemin pour notre raccourci. Ce chemin est semblable à un chemin de fichier : "<WINDOWTYPE>/Category1/Category2/.../Action" où *WINDOWTYPE* est un identifiant spécifique à chaque application, pour notre application, nous utiliserons *EditeurGTK*, ensuite la documentation de **GTK+** conseille, pour les éléments du menu, d'utiliser son chemin, par exemple pour l'élément Nouveau : "Fichier/Nouveau" ce qui nous donne : "<EditeurGTK>/Fichier/Nouveau". Comme il va être nécessaire de reprendre ces chemins lors de la création des éléments du menu, il est préférable d'en faire des constantes :

raccourcis.h
```
#define ACCEL_PATH_NEW "<EditeurGTK>/Fichier/Nouveau"
#define ACCEL_PATH_OPEN "<EditeurGTK>/Fichier/Ouvrir"
#define ACCEL_PATH_SAVE "<EditeurGTK>/Fichier/Enregistrer"
#define ACCEL_PATH_SAVEAS "<EditeurGTK>/Fichier/Enregistrer sous"
#define ACCEL_PATH_CLOSE "<EditeurGTK>/Fichier/Fermer"
#define ACCEL_PATH_QUIT "<EditeurGTK>/Fichier/Quitter"
```

Avant de créer nos raccourcis, il faut régler un problème (sur ce point je trouve que **GTK+** est mal fait), en effet il est précisé que la fonction *callback* pour les raccourcis doit avoir la signature suivante :

```
gboolean (*GtkAccelGroupActivate) (GtkAccelGroup *accel_group, GObject *acceleratable, guint
 keyval, GdkModifierType modifier);
```

Alors que nous avons des fonctions de la forme :

```
void callback (GtkWidget *p_widget, gpointer user_data);
```

On est donc obligé de créer des fonctions de type *GtkAccelGroupActivate* qui vont se charger d'appeler nos fonctions *callback* :

raccourcis.c
```
static gboolean accel_new (GtkAccelGroup *accel_group, GObject *acceleratable, guint keyval,
 GdkModifierType modifier, gpointer user_data)
{
  cb_new (NULL, user_data);
  return TRUE;
}
```

Notre fonction n'a pas la même signature que les GtkAccelGroupActivate puisque g_cclosure_new précise qu'il appelle la fonction callback ainsi créée avec user_data comme dernier paramètre.

Maintenant que le problème est résolu, créons nos raccourcis :

raccourcis.c
```
     accelerator_new (p_accel_group, "<Control>N", ACCEL_PATH_NEW, G_CALLBACK (accel_new), user_data);
     accelerator_new (p_accel_group, "<Control>O", ACCEL_PATH_OPEN, G_CALLBACK (accel_open),
    user_data);
     accelerator_new (p_accel_group, "<Control>S", ACCEL_PATH_SAVE, G_CALLBACK (accel_save),
    user_data);
     accelerator_new (p_accel_group, "<Control><Shift>S", ACCEL_PATH_SAVEAS, G_CALLBACK
    (accel_saveas), user_data);
     accelerator_new (p_accel_group, "<Control>W", ACCEL_PATH_CLOSE, G_CALLBACK (accel_close),
    user_data);
     accelerator_new (p_accel_group, "<Control>Q", ACCEL_PATH_QUIT, G_CALLBACK (accel_quit),
    user_data);
```

Pour finir, il faut ajouter le *GtkAccelGroup* à notre fenêtre principale :

raccourcis.c
```
     gtk_window_add_accel_group (docs.p_main_window, p_accel_group);
```

Voilà, nous en avons fini avec ce fichier, maintenant il faut revenir à *menu.c* pour associer les raccouris aux éléments du menu. Il nous suffit de modifier notre constructeur de *GtkMenuItem* pour qu'il prenne en argument le *accel_path* :

menu.c
```
    static void menu_item_new (GtkMenu *p_menu, const gchar *title, const gchar *accel_path, GCallback
     callback, gpointer user_data)
    {
      /* ... */
      gtk_menu_item_set_accel_path (GTK_MENU_ITEM (p_menu_item), accel_path);
    }
```

Et d'utiliser nos constantes lors de l'appel à la fonction *meni_item_new* :

menu.c
```
       menu_item_new (GTK_MENU (p_menu), "_Nouveau", ACCEL_PATH_NEW, G_CALLBACK (cb_new), user_data);
```

XIV-C - Code source

chapitre14.zip

XV - Messages d'erreur

XV-A - Aperçu

Cliquez pour agrandir

XV-B - Amélioration de nos fonctions d'affichage d'erreur

Jusqu'à présent les messages d'erreurs étaient affichés à l'aide de la fonction *printf* mais cela oblige l'utilisateur à lancer le programme dans une console, pas très conviviale. Encore une fois **GTK+** vient à notre secours en proposant une classe *GtkMessageDialog* qui nous permet d'afficher un message simplement, sans avoir à créer une boîte de dialogue en partant de zéro.

Voici la fonction qui nous permet de créer notre boîte de dialogue :

```
GtkWidget *gtk_message_dialog_new (GtkWindow *parent, GtkDialogFlags flags, GtkMessageType type,
 GtkButtonsType buttons,
                                   const gchar *message_format, ...);
```

Donc nous avons besoin de notre fenêtre principale, pour cela il suffit d'inclure *document.h*, comme l'utilisateur doit d'abord valider notre message avant de continuer à utiliser l'éditeur, nous allons la rendre modale grâce à la constante *GTK_DIALOG_MODAL*. Ensuite vient le type de message, cela va dépendre de la fonction appelée (*GTK_MESSAGE_INFO*, *GTK_MESSAGE_WARNING* ou *GTK_MESSAGE_ERROR*). Le seul bouton dont l'utilisateur a besoin est le bouton Ok pour fermer la boîte de dialogue (*GTK_BUTTONS_OK*) et pour finir, la fonction attend le message à afficher (sous la même forme que la fonction *printf*).

Comme seul le type de message et le message va changer selon la fonction *print_** appelée, une nouvelle fonction est la bienvenue :

error.c
```
static void print_message (GtkMessageType type, const gchar *format, va_list va)
{
  gchar *message = NULL;
  GtkWidget *p_dialog = NULL;

  message = g_strdup_vprintf (format, va);
  p_dialog = gtk_message_dialog_new (docs.p_main_window, GTK_DIALOG_MODAL, type, GTK_BUTTONS_OK,
 message);
  g_free (message), message = NULL;
  gtk_dialog_run (GTK_DIALOG (p_dialog));
  gtk_widget_destroy (p_dialog);
}
```

Les fonctions de la famille de *g_strdup_printf* sont extrêmement intéressantes puisqu'elles permettent de créer une nouvelle chaîne de caractères en utilisant la puissance des fonctions de la famille de *printf* (Voici un exemple d'implémentation de ce genre de fonction : **Créer une chaîne de caractères formatée**).

Il nous reste plus qu'à modifier nos trois fonctions d'affichage de message, voici l'exemple pour *print_info* :

error.c
```
void print_info (char *format, ...)
{
  va_list va;
```

error.c

```
    va_start (va, format);
    print_message (GTK_MESSAGE_INFO, format, va);
  }
```

En C99, avec les macro à nombres variables, nous pourrions remplacer nos fonctions par des macro :

```
#define print_info(chat *format, ...) print_message (GTK_MESSAGE_INFO, (format), __VA_ARGS__)
```

XV-C - Code source

chapitre15.zip

XVI - Ouvrir plusieurs fichiers en même temps

XVI-A - Aperçu

Cliquez pour agrandir

XVI-B - Mise en place des onglets

Actuellement, nous ne pouvons ouvrir qu'un seul fichier à la fois. Les éditeurs de texte avancés proposent généralement la possibilité d'ouvrir plusieurs documents simultanément grâce à un système d'onglets. C'est ce que nous allons mettre en place grâce au *widget GtkNotebook*.

A la place du *GtkTextView*, nous créons un *GtkNotebook* et comme nous allons avoir besoin de modifier de *widget* (ajout/suppression de pages), nous gardons un pointeur dessus dans notre structure globale :

main.c
```
    /* Creation de la page d'onglets */
    {
      GtkWidget *p_notebook = NULL;

      p_notebook = gtk_notebook_new ();
      gtk_container_add (GTK_CONTAINER (p_main_box), p_notebook);
      /* ... */
      docs.p_notebook = GTK_NOTEBOOK (p_notebook);
    }
```

Donc à l'ouverture de l'éditeur, aucun onglet est ouvert, ils seront ajoutés lorsque l'utilisateur crée un document ou en ouvre un. Par chance (ou grâce à l'ingéniosité de l'auteur de ce document, je vous laisse choisir :D), l'ajout d'une nouvelle page se fait uniquement dans la fonction *cb_new*. Nous avons anticipé la mise en place d'onglet au début de ce tutoriel en créant la structure *docs_t* dont seul le champs *actif* était utilisé, maintenant, il faut ajouter chaque document ouvert à la *GList* :

callback.c
```
  void cb_new (GtkWidget *p_widget, gpointer user_data)
  {
    document_t *nouveau = NULL;

    nouveau = g_malloc (sizeof (*nouveau));
    nouveau->chemin = NULL;
    /* Le document vient d'etre ouvert, il n'est donc pas modifie */
    nouveau->sauve = TRUE;
    docs.tous = g_list_append (docs.tous, nouveau);
    {
      gint index = 0;
      GtkWidget *p_scrolled_window = NULL;

      p_scrolled_window = gtk_scrolled_window_new (NULL, NULL);
      gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (p_scrolled_window), GTK_POLICY_AUTOMATIC,
   GTK_POLICY_AUTOMATIC);
      nouveau->p_text_view = GTK_TEXT_VIEW (gtk_text_view_new ());
      {
        GtkTextBuffer *p_buffer = NULL;

        p_buffer = gtk_text_view_get_buffer (nouveau->p_text_view);
        g_signal_connect (G_OBJECT (p_buffer), "changed", G_CALLBACK (cb_modifie), NULL);
      }
      gtk_container_add (GTK_CONTAINER (p_scrolled_window), GTK_WIDGET (nouveau->p_text_view));
```

callback.c
```
        index = gtk_notebook_append_page (docs.p_notebook, p_scrolled_window, GTK_WIDGET (gtk_label_new
    ("Nouveau document")));
        gtk_widget_show_all (p_scrolled_window);
        gtk_notebook_set_current_page (docs.p_notebook, index);
     }

     /* parametres inutilises */
     (void)p_widget;
     (void)user_data;
   }
```

Première chose intéressante, il n'est plus nécessaire de fermer le document pour en ouvrir un autre. Ensuite, plutôt que de modifier le champ *actif*, on crée un nouveau document que l'on ajoute à la *GList*. Le *GtkTextView* est créé comme précédemment mis à part qu'il est ajouté à un nouvel onglet que l'on crée grâce à la fonction :

```
gint gtk_notebook_append_page (GtkNotebook *notebook, GtkWidget *child, GtkWidget *tab_label);
```

Qui a besoin du *widget* enfant (il s'agit du *GtkScrolledWindow* contenant le *GtkTextView*) et d'un second *widget* qui sera affiché dans l'onglet (nous laissons **GTK+** mettre un texte par défaut, nous verrons plus loin comment améliorer cela).

Une fois l'onglet créé, *gtk_notebook_append_page* nous retourne la position de la nouvelle page créée qui va nous servir à rendre la page active grâce à la fonction :

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

XVI-C - Changement de page

Pour pouvoir mettre à jour le document actif lorsque l'utilisateur navigue entre les différents onglets, il suffit d'intercepter le signal *switch-page* :

main.c
```
g_signal_connect (G_OBJECT (p_notebook), "switch-page", G_CALLBACK (cb_page_change), NULL);
```

La fonction *cb_page_change* récupére la position de la page courante et fait pointer *actif* vers la structure *document_t* correspondante stockée dans la *GList* :

callback.c
```
void cb_page_change (GtkNotebook *notebook, GtkNotebookPage *page, guint page_num, gpointer
 user_data)
{
  docs.actif = g_list_nth_data (docs.tous, page_num);

  /* parametres inutilises */
  (void)notebook;
  (void)user_data;
}
```

Comme **GTK+** fait bien les choses, la fonction *gtk_notebook_set_current_page* que nous appelons lors de la création d'un document émet ce signal, donc pas besoin de recopier ce code dans *cb_new*.

XVI-D - Fermer un onglet

La dernière fonction qui nécessite quelques modifications, est *cb_close*, il faut maintenant supprimer le document de la *GList*, libérer la mémoire et supprimer l'onglet :

callback.c
```
void cb_close (GtkWidget *p_widget, gpointer user_data)
{
  /* Avant de fermer, il faut verifier qu'un document a bien ete ouvert */
  if (docs.actif)
  {
    /* ... */
    {
      docs.tous = g_list_remove (docs.tous, docs.actif);
      g_free (docs.actif->chemin), docs.actif->chemin = NULL;
      g_free (docs.actif), docs.actif = NULL;
      gtk_notebook_remove_page (docs.p_notebook, gtk_notebook_get_current_page (docs.p_notebook));
      if (gtk_notebook_get_n_pages (docs.p_notebook) > 0)
      {
        docs.actif = g_list_nth_data (docs.tous, gtk_notebook_get_current_page (docs.p_notebook));
      }
      else
      {
        docs.actif = NULL;
      }
    }
  }
  else
  {
    print_warning ("Aucun document ouvert");
  }

  /* parametres inutilises */
  (void)p_widget;
  (void)user_data;
}
```

Il reste plus qu'à supprimer l'appel à la fonction *gtk_widget_set_sensitive* dans la fonction *open_file*, maintenant devenu inutile.

Bizarrement la suppression d'un onglet n'émet pas de signal switch-page, nous devons le faire manuellement.

XVI-E - Fermer tous les onglets

Maintenant que nous pouvons ouvrir plusieurs documents en même temps, il est nécessaire de les fermer tous lorsqu'on quitte le programme.

J'ai essayé plusieurs méthodes avant de trouver une méthode convenable. Contrairement à ce que l'on pourrait penser, il ne suffit pas d'utiliser la fonction *g_list_foreach*, qui permet de parcourir l'ensemble d'une liste chaînée, et de fermer les documents un à un. Le problème vient des documents non sauvegardés, puisque l'utilisateur peut à tout moment décider d'annuler l'enregistrement d'un document ce qui nous oblige à annuler la fermeture de l'application.

Le principe que j'ai choisi d'adopter consiste à boucler tant que tous les documents ne sont pas fermés (dans ce cas *docs.actif* vaut NULL) et de demander la fermeture du premier onglet (puisque le numéro du dernier change à chaque itération). Si l'utilisateur choisit d'annuler l'enregistrement, dans ce cas l'onglet n'est pas fermé et le nombre de documents ouverts est le même avant et après l'appel à *cb_close*, nous mettons alors fin à la boucle et signalons l'annulation en retournant FALSE, si tous les documents ont bien été fermés, la fonction renvoit TRUE :

```
static gboolean close_all (void)
```

```
{
  gboolean ret = TRUE;

  while (docs.actif)
  {
    gint tmp = gtk_notebook_get_n_pages (docs.p_notebook);

    gtk_notebook_set_current_page (docs.p_notebook, 0);
    cb_close (NULL, NULL);
    if (gtk_notebook_get_n_pages (docs.p_notebook) >= tmp)
    {
      ret = FALSE;
      break;
    }
  }
  return ret;
}
```

Et la fonction *cb_quit* devient :

```
void cb_quit (GtkWidget *p_widget, gpointer user_data)
{
  if (close_all ())
  {
    g_free (docs.dir_name), docs.dir_name = NULL;
    gtk_main_quit();
  }

  /* parametres inutilises */
  (void)p_widget;
  (void)user_data;
}
```

XVI-F - Modifier le titre de la page

Pour plus d'esthétisme, nous allons modifier les titres des onglets. Pour les nouveaux documents, nous afficherons un texte par défaut (*Nouveau document* par exemple), une fois enregistré nous afficherons le nom du fichier (sans le chemin pour faire court). Et lorsque le document a été modifié depuis la dernière sauvegarde, nous ajouterons un petit astérisque à côté du titre.

Les bases étant posées, nous allons commencer par construire notre titre :

callback.c
```
static void set_title (void)
{
  if (docs.actif)
  {
    gchar *title = NULL;
    gchar *tmp = NULL;

    if (docs.actif->chemin)
    {
      tmp = g_path_get_basename (docs.actif->chemin);
    }
    else
    {
      tmp = g_strdup ("Nouveau document");
    }
    if (docs.actif->sauve)
    {
      title = g_strdup (tmp);
    }
```

callback.c
```
      else
      {
        title = g_strdup_printf ("%s *", tmp);
      }
      g_free (tmp), tmp = NULL;
      /* ... */
      g_free (title), title = NULL;
    }
  }
```

Quelques lignes de code simplifiées grâce aux fonctions de la **glib**. Nous obtenons donc notre titre dans la variable *titre* qui nous reste plus qu'à insérer dans l'onglet :

callback.c
```
      {
        gint index = 0;
        GtkWidget *p_child = NULL;
        GtkLabel *p_label = NULL;

        index = gtk_notebook_get_current_page (docs.p_notebook);
        p_child = gtk_notebook_get_nth_page (docs.p_notebook, index);
        p_label = GTK_LABEL (gtk_notebook_get_tab_label (docs.p_notebook, p_child));
        gtk_label_set_text (p_label, title);
      }
```

Cela peut paraître bizarre mais modifier le titre d'un onglet n'est pas trivial, il faut commencer par récupérer le numéro de la page en cours pour obtenir le *widget* qui est affiché dans l'onglet (car nous sommes libre de mettre le *widget* de notre choix), et comme dans notre cas, c'est un simple *GtkLabel*, on utilise la fonction *gtk_label_set_text* pour mettre à jour le titre. Pour finir, il faut faire appel à la fonction *set_title* lorsqu'on crée, ouvre, sauvegarde ou modifie un document, c'est à dire respectivement dans les fonctions *cb_new*, *open_file*, *cb_save* et *cb_modifie*.

Et voilà, c'est tout ! Moyennant quelques efforts de réflexion au début, le changement entre un éditeur simple et multi-documents est extrêmement simple et a même simplifié notre code : par exemple, pour la création du menu, nous n'avons plus besoin du paramètre *user_data* (pour simplifier, et au cas où nous en aurions de nouveau besoin, nous passons la valeur NULL).

XVI-G - Code source

chapitre16.zip

XVII - Afficher l'arborescence du disque

XVII-A - Aperçu

Cliquez pour agrandir

XVII-B - Préparons le terrain

Nous allons avoir besoin d'ajouter un *GtkTreeView* à côté du *GtkTextView*. La première idée qui vient à l'esprit est de créer un *GtkHBox* dans la boîte principale. Mais pour varier les plaisirs, nous allons utiliser un nouveau type de conteneur : les *GtkPaned*, ils permettent de séparer une zone en deux parties séparées par une barre mobile.

Nous créons donc un *GtkHPaned* (la séparation se fait dans le sens horizontal, à l'opposé des *GtkVPaned*) :

main.c
```
GtkWidget *p_hpaned = NULL;

/* ... */
p_hpaned = gtk_hpaned_new ();
gtk_box_pack_start (GTK_BOX (p_main_box), p_hpaned, TRUE, TRUE, 0);
```

Et plutôt que d'afficher la *GtkNotebook* dans la boîte principale, nous l'affichons maintenant dans la seconde partie de notre nouveau containeur :

main.c
```
gtk_paned_add2 (GTK_PANED (p_hpaned), p_notebook);
```

XVII-C - Création d'un GtkTreeView

Les *GtkTreeView* sont sûrement les *widgets* les plus difficiles à maîtriser. Ceci est due au fait que la gestion du contenu et de l'affichage est séparé en deux *widgets* et qu'il est possible d'afficher une simple liste ou un arbre :

- *GtkListStore* et *GtkTreeStore* pour stocker respectivement, le contenu d'une liste et d'un arbre
- *GtkTreeView* pour afficher le contenu des Gtk*Store.

Nous avons donc le choix entre afficher le contenu du répertoire courant ou afficher toute l'arborescence du disque. Nous opterons pour la première solution car lister tout le contenu du disque serait bien trop long (surtout de nos jours où un disque dur fait plusieurs dizaines de GO). Mais pour ne pas se limiter au répertoire où se trouve notre programme (ce qui serait gênant s'il se trouve dans /usr/bin), nous allons aussi lister les répertoires présents pour permettre à l'utilisateur de naviguer dans l'arborescence.

Pour résumer nos objectifs : nous voulons créer un *GtkTreeView* qui affichera un *GtkListStore* contenant le nom des fichiers et dossiers présents dans un répertoire donné. Lorsque l'utilisateur clique sur un nom représentant un fichier, on l'ouvre, s'il s'agit d'un dossier, on réactualise le *GtkListStore* avec le contenu de ce nouveau répertoire.

Y a plus qu'a...

XVII-C-1 - Création du magasin

main.c
```
	p_list_store = gtk_list_store_new (2, GDK_TYPE_PIXBUF, G_TYPE_STRING);
	docs.p_list_store = p_list_store;
	docs.dir_name = g_strdup (g_get_home_dir ());
	dir_list ();
```

Qui a osé dire qu'il s'agissait de la partie la plus difficile ? Vous l'aurez compris, tout le code se trouve dans la fonction *dir_list* que nous verrons plus tard. Pour commencer, on construit notre *GtkListStore* en précisant le nombre de colonnes souhaité suivi de leurs types. Nous souhaitons deux colonnes, la première contiendra un *GdkPixbuf* (une image) pour différencier les dossiers des fichiers, et la seconde le nom du fichier.

Ensuite, nous sauvegardons notre *GtkListStrore* et le chemin du dossier à afficher (la fonction *g_get_home_dir* nous permet de connaître le dossier de l'utilisateur) dans notre structure globale car nous en avons besoin dans plusieurs fonctions *callback* par la suite.

Maintenant passons au remplissage du magasin. Comme nous serons amenés à rafraîchir le contenu de ce dernier, on commence par le vider :

callback.c
```
	gtk_list_store_clear (docs.p_list_store);
```

Pour lister le contenu du répertoire, nous allons utiliser la **glib**. Après avoir ouvert le répertoire, il nous suffit d'appeler la fonction *g_dir_read_name* qui à chaque appel nous renvoie le fichier (9) suivant puis NULL lorsque tout le répertoire a été parcouru :

callback.c
```
void dir_list (void)
{
  GDir *dir = NULL;

  dir = g_dir_open (docs.dir_name, 0, NULL);
  if (dir)
  {
    const gchar *read_name = NULL;

    /* ... */
    while ((read_name = g_dir_read_name (dir)))
    {
      /* ... */
    }
    g_dir_close (dir), dir = NULL;
  }
}
```

Pour chaque fichier, il faut commencer par reconstruire son chemin complet :

callback.c
```
	  gchar *file_name = NULL;

	  file_name = g_build_path (G_DIR_SEPARATOR_S, docs.dir_name, read_name, NULL);
```

La fonction *g_build_path* concatène l'ensemble de ses arguments, sauf le premier qui est le séparateur utilisé. Maintenant que nous avons notre nom complet, nous pouvons tester si notre fichiers est un dossier ou pas :

callback.c
```
	GdkPixbuf *p_file_image = NULL;
	  /* ... */
```

callback.c
```
        if (g_file_test (file_name, G_FILE_TEST_IS_DIR))
        {
          p_file_image = gdk_pixbuf_new_from_file ("dossier.png", NULL);
        }
        else
        {
          p_file_image = gdk_pixbuf_new_from_file ("fichier.png", NULL);
        }
```

La fonction permet de tester différentes propriétés relatives aux fichiers :

```
typedef enum
{
  G_FILE_TEST_IS_REGULAR    = 1 << 0, /* TRUE si le fichier est un fichier standard (ni un lien
 symbolique ni un dossier) */
  G_FILE_TEST_IS_SYMLINK    = 1 << 1, /* TRUE si le fichier est un lien symbolique */
  G_FILE_TEST_IS_DIR        = 1 << 2, /* TRUE si le fichier est un dossier */
  G_FILE_TEST_IS_EXECUTABLE = 1 << 3, /* TRUE si le fichier est un executable */
  G_FILE_TEST_EXISTS        = 1 << 4  /* TRUE si le fichier existe */
} GFileTest;
```

Donc selon le type de fichier, nous chargeons l'image approprié dans un *GdkPixbuf*. Le second paramètre de la fonction *gdk_pixbuf_new_from_file* permet de récupérer plus d'information lorsqu'une erreur survient.

Pour finir, il nous reste plus qu'à ajouter une nouvelle colonne dans le *GtkListStore*. Ceci se réalise en deux étapes. La première consiste à ajouter une colonne :

callback.c
```
      GtkTreeIter iter;
/* ... */
      gtk_list_store_append (docs.p_list_store, &iter);
```

Comme pour le *GtkTextView*, nous avons besoin d'un itérateur qui va nous permettre de remplir cette colonne à l'aide d'une seconde fonction :

callback.c
```
        gtk_list_store_set (docs.p_list_store, &iter, 0, p_file_image, 1, read_name, -1);
```

Le premier paramètre est bien sûr notre magasin, ensuite il s'agit de l'itérateur symbolisant la colonne nouvellement créée et enfin des couples numéro de colonne/donnée qui doivent correspondre au nombre et type précisé lors de la création du *GkListStore*.

En plus de cela, nous ajoutons aussi un dossier "..", puisque la fonction *g_dir_read_name* ne le liste pas, pour permettre à l'utilisateur de remonter dans l'arborescence.

XVII-C-2 - Affichage de l'arborescence

Voilà notre *GtkListStore* est prêt ! Maintenant il nous faut associer ce dernier au *GtkTreeView*. Ceci n'a besoin d'être fait qu'une seule fois lors de la création du *widget* :

main.c
```
      p_tree_view = gtk_tree_view_new_with_model (GTK_TREE_MODEL (p_list_store));
```

GtkTreeModel est une interface utilisée par *GtkTreeView*, la classe *GtkListStore* implémentant cette interface, il est possible de réaliser un transtypage.

Si l'histoire s'arrêterai là, créer un *GtkTextView* sera plutôt simple. Hélas nous devons créer les colonnes dans le *GtkTextView* et les faire correspondre à celles du magasin.

Le nombre d'éléments affichables par une cellule d'un *GtkTreeView* étant varié, il faut commencer par créer un *GtkCellRenderer* qui peut être de différents types :

- *GtkCellRendererText* : pour afficher du texte
- *GtkCellRendererPixbuf* : pour les images
- *GtkCellRendererProgress* : permet d'ajouter une barre de progression
- *GtkCellRendererToggle* : affiche une case à cocher.

Dans notre cas, nous avons besoin que des deux premiers. Voici le code pour la première colonne qui contiendra l'image :

main.c
```
        GtkCellRenderer *p_renderer = NULL;
    /* ... */
        p_renderer = gtk_cell_renderer_pixbuf_new ();
```

Une fois la cellule créée, il faut créer la colonne à proprement parlée grâce à la fonction :

```
    GtkTreeViewColumn* gtk_tree_view_column_new_with_attributes
                                            (const gchar *title,
                                             GtkCellRenderer *cell,
                                             ...);
```

Après le titre de la colonne et le *GtkCellRenderer*, la fonction attend une chaîne de caractères qui diffère selon le type de *GtkCellRenderer* suivi du numéro de la colonne. Comme il est possible de passer plusieurs couples identifiant/numéro de colonne, nous terminons la liste par NULL.

Et pour terminer, on ajoute la colonne au *GtkTextView* :

main.c
```
        gtk_tree_view_append_column (GTK_TREE_VIEW (p_tree_view), p_column);
```

La démarche est identique pour la seconde colonne :

main.c
```
        p_renderer = gtk_cell_renderer_text_new ();
        p_column = gtk_tree_view_column_new_with_attributes (NULL, p_renderer, "text", 1, NULL);
        gtk_tree_view_append_column (GTK_TREE_VIEW (p_tree_view), p_column);
```

Et comme nous n'avons pas besoin des en-têtes de colonnes, nous les cachons :

main.c
```
        gtk_tree_view_set_headers_visible (GTK_TREE_VIEW (p_tree_view), FALSE);
```

Je suis passé rapidement sur cette partie car la documentation officielle n'est pas très complète à ce sujet et le rôle des fonctions n'est pas très claire.

XVII-C-3 - Sélectionner un fichier

Nous arrivons à afficher le contenu d'un dossier, il nous reste plus qu'à réagir à la sélection d'une colonne. Vous commencez à être habitué, il faut intercepter le signal *row-activated* du *GtkTextView* :

main.c
```
g_signal_connect (G_OBJECT (p_tree_view), "row-activated", G_CALLBACK (cb_select), NULL);
```

La fonction *callback* correspondante est différente de ce qu'on a l'habitude de voir :

```
void cb_select (GtkTreeView *p_tree_view, GtkTreePath *arg1, GtkTreeViewColumn *arg2, gpointer
 user_data)
{
  /* ... */
}
```

Ces arguments vont nous permettrent de retrouver la cellule sélectionnée. La méthode est comparable à celle que l'on utilise pour les *GtkTexView*, on commence par récupérer notre magasin sous forme de *GtkTreeModel* comme nous pourrions le faire avec un *GtkTextBuffer* :

```
GtkTreeModel *p_tree_model = NULL;

p_tree_model = gtk_tree_view_get_model (p_tree_view);
```

Ensuite à l'aide du *GtkTreePath*, qui est une représentation du chemin pour accéder à l'élément sélectionné, nous récupérons un *GtkTreeIter* :

```
GtkTreeIter iter;
/* ... */
gtk_tree_model_get_iter (p_tree_model, &iter, arg1);
```

Et pour finir, on récupère le contenu de la seconde colonne grâce au *GtkTreeIter* :

```
gchar *str = NULL;
/* ... */
gtk_tree_model_get (p_tree_model, &iter, 1, &str, -1);
```

Nous pourrions récupérer plusieurs colonnes en même temps en ajoutant d'autre couple numéro de colonne/pointeur sur un type de variable compatible avec ce que nous avons stocké dans le *GtkListStore*.

Voilà c'est la fin de la partie floue (je m'en excuse mais la documentation n'est pas très complète à se sujet, il faut donc faire des essais en tâtonnant jusqu'à se que cela fonctionne sans forcément en connaître les raisons :().

Maintenant que nous avons notre nom de fichier, il faut retrouver son chemin complet et tester s'il s'agit d'un dossier ou non. Ceci a déjà été vu lors de la création du *GtkListStore* :

```
gchar *file_name = NULL;
/* ... */
file_name = g_build_path (G_DIR_SEPARATOR_S, docs.dir_name, str, NULL);
g_free (str), str = NULL;
if (g_file_test (file_name, G_FILE_TEST_IS_DIR))
{
  /* ... */
}
else
```

```
    {
      /* ... */
    }
```

Commençons par le plus difficile : dans le cas où notre fichier est un dossier, il suffit de remplacer le nom du dossier à afficher et de rafraîchir le *GtkListStore* en faisant appel à la fonction *dir_list* :

```
        g_free (docs.dir_name), docs.dir_name = NULL;
        docs.dir_name = file_name;
        dir_list ();
```

Difficile de faire plus simple ! Pour ouvrir un fichier, il suffit de faire appel à la fonction *open_file* :

```
        open_file (file_name);
```

XVII-D - Code source

chapitre17.zip

XVIII - Notre signature

XVIII-A - Aperçu

Cliquez pour agrandir

XVIII-B - Boîte A propos

Nous allons terminer cette initiation par un petit ajout qui va finir notre application : une boîte de dialogue A propos. Depuis la version 2.6 de **GTK+**, il existe un *widget* qui va nous simplifier la vie : *GtkAboutDialog*.

Son utilisation est plutôt simple, il suffit de créer le *widget*, puis un certain nombre de fonctions permettent de renseigner les informations concernant le programme (auteur, version licence...) puis on affiche la boîte de dialogue :

```
void cb_about (GtkWidget *p_widget, gpointer user_data)
{
  GtkWidget *p_about_dialog = NULL;

  p_about_dialog = gtk_about_dialog_new ();
  gtk_about_dialog_set_version (GTK_ABOUT_DIALOG (p_about_dialog), "1.0");
  gtk_about_dialog_set_name (GTK_ABOUT_DIALOG (p_about_dialog), "Editeur de texte");

  {
    const gchar *authors[2] = {"gege2061", NULL};

    gtk_about_dialog_set_authors (GTK_ABOUT_DIALOG (p_about_dialog), authors);
  }
  {
    gchar *contents = NULL;

    if (g_file_get_contents ("COPYING", &contents, NULL, NULL))
    {
      gchar *utf8 = NULL;

      utf8 = g_locale_to_utf8 (contents, -1, NULL, NULL, NULL);
      g_free (contents), contents = NULL;
      gtk_about_dialog_set_license (GTK_ABOUT_DIALOG (p_about_dialog), utf8);
      g_free (utf8), utf8 = NULL;
    }
  }
  gtk_about_dialog_set_website (GTK_ABOUT_DIALOG (p_about_dialog),
 "http://nicolasj.developpez.com/");
  {
    GdkPixbuf *p_logo = NULL;

    p_logo = gdk_pixbuf_new_from_file ("logo.png", NULL);
    gtk_about_dialog_set_logo (GTK_ABOUT_DIALOG (p_about_dialog), p_logo);
  }
  gtk_dialog_run (GTK_DIALOG (p_about_dialog));

  /* parametres inutilises */
  (void)p_widget;
  (void)user_data;
}
```

Voilà rien de bien compliqué mais le résultat obtenu est plutôt sympathique.

Bizarrement pour la fonction gtk_about_dialog_set_authors le tableau doit être déclaré comme constant. Un tableau non constant provoque une erreur de compilation !

XVIII-C - Code source

chapitre18.zip

XIX - Conclusion

XIX-A - C'est déjà fini ?

Eh oui c'est la fin de ce tutoriel ! Mais si vous avez pris autant de plaisir que moi à lire et surtout commencer à maîtriser la puissance de **GTK+**, alors ne vous inquiétez pas notre éditeur n'en est qu'a la version 1.0. Les prochaines versions ne sont pas prévues pour la correction des bugs (enfin j'espère), mais plutôt ajouter de nouvelles fonctionnalités :D Voici un aperçu des possibilités de **GTK+** que je n'ai pas abordé ici mais qui pourront faire l'objet de tutoriels :

- La création d'un écran de démarrage
- Utilisation du *widget GtkSourceView* pour la coloration syntaxique
- Le drag'n'drop
- Une meilleure gestion des erreurs grâce au *GError*
- Et plein d'autres choses que je ne connais pas encore...

Je vous l'ai déjà dit, mais je préfère le répéter, la documentation de l'API **GTK+** est votre première source d'information, n'hésitez pas à passer quelques minutes à chercher une fonction avant de recréer la roue et surtout faites des essais, c'est comme cela que l'on apprend (j'ai découvert énormément de fonctionnalités en écrivant ce tutoriel).

Si malgré cela vous avez des difficultés lors de vos développements avec **GTK+**, les membres du **forum C** seront, j'en suis sûr, ravis de vous venir en aide ;)

XIX-B - Remerciements

Merci à **loka**, **fearyourself** et à **Miles** pour leur relecture attentive de cet article.

1 : **Widget** est l'acronyme de ***W**indows, **I**cons, **D**ialog box, **G**raphics **E**xtensions, **T**rack ball*, plus souvent remplacé par ***WI**n**D**ows gad**GET***
2 : Le C n'est certes pas un langage fait prévu pour la POO mais en poussant à l'extrême la notion de type abstrait de donnée (TAD), **GTK+** offre des possibilités proche de la POO.
3 : ce que je vous conseille de faire pour voir le problème, n'oubliez pas de créer une méthode *cb_open* sur le même modèle que *cb_quit* pour pouvoir compiler
4 : La **glib** contient de nombreuses fonctions fortes utiles, il m'est souvent arrivé de coder une fonction qui était en fait présente dans la **glib**, n'hésitez pas à perdre quelques minutes à passer sa documentation en revue pour éviter une perte de temps.
5 : Oui ça ressemble, de loin, aux itérateurs du C# pour ceux qui connaissent.
6 : Il va falloir vous y faire, à moins d'être un surdoué, il est impossible de connaître l'API par coeur alors prenez le réflexe d'avoir toujours la documentation à portée de main.
7 : Une boîte de dialogue modale empêche l'utilisateur d'interagir avec sa fenêtre parent
8 : Nous n'aurions pas eu ce problème si nous commencions par créer tous les *widgets* puis les intégrions à la fenêtre. Le problème avec cette méthode c'est que l'on a besoin des variables désignant les *widgets* jusqu'à la fin de la fonction, ce qui nous empêcherait de découper le code sous forme de blocs dans lesquels nous créons chaque élément.
9 : Ici fichier est à prendre au sens Unix du terme, c'est à dire que tout est fichier.