Tutoriel d'introduction à Gtkmm

Ce tutoriel est inspiré du tutoriel de X. Garreau (http://www.xgarreau.org/).

1 Introduction

Gtkmm est au C++ ce que le Gtk+ est au C en terme de fonctionnalités offertes. Gtkmm permet de créer des interfaces graphiques comme Gtk+ peut le faire. Gtkmm est un ensemble de ce qu'on appelle des "wrappers", des "englobeurs" en vilain français. Le but d'un wrapper est de créer une interface pour un langage dont l'implémentation est réalisée à partie d'un autre langage. Quand vous coderez à l'aide de Gtkmm, vous manipulerez des instances de classes de façon habituelle. Pourtant ces classes utilisent en fait des appels à une librairie en C. Il s'agit bien sûr de Gtk+.

Pourquoi ne pas utiliser Gtk+ directement ? Concentrez vous sur l'aspect objet de votre projet, d'autres ont pris la peine de créer des classes. S'il vous démange d'utiliser Gtk+ dans du code C++, vous pouvez toutefois le faire.

Dans ce tutoriel, on va s'intéresser à une application simple : pour faire un peu plus évolué qu'un simple « Hello World ! », elle sera composée d'un texte explicatif, d'une zone pour saisir du texte et d'un bouton. Dans la zone de texte, on saisira son nom et lors du clic sur le bouton, un dialogue vous dira Bonjour.

Commençons par les choses simples... Pour développer un programme utilisant Gtkmm, vous devez inclure une ligne `#include <Gtkmm.h>` dans votre code source. Ça c'est la méthode brutale. Vous incluez ainsi toutes les entêtes de Gtkmm... Vous pouvez faire les choses plus finement en n'incluant que les éléments qui vous servent dans votre programme. À titre d'exemple, notre application nécessite les inclusions suivantes :

```
#include <Gtkmm/main.h>
#include <Gtkmm/window.h>
#include <Gtkmm/box.h>
#include <Gtkmm/button.h>
#include <Gtkmm/label.h>
#include <Gtkmm/entry.h>
```

La première ligne est obligatoire puisqu'elle inclut la définition la classe `Gtk::Main` permettant de lancer la boucle Gtk principale. Les autres includes on des noms explicites, ils nous permettent l'utilisation de fenêtres, boîtes de rangement, boutons, texte statique et zone de texte éditable.

La librairie Gtkmm s'appuie sur `glibmm`. Le contenu des librairies sont réparties dans plusieurs espaces de noms, `Gtk`, `Gdk`, `Glib`, `Atk` et `Pango` sont les cinq principaux. Il est déconseillé d'utiliser les directives `using namespace` mais plutôt de spécifier les espaces de nom lorsque c'est nécessaire.

Lorsqu'on développe avec un toolkit tel que Qt ou Gtkmm, les différents widgets sont représentés par des classes. Comment savoir si on doit instancier un objet à partir d'une classe du toolkit ou créer sa propre classe qui hérite de ce widget ? Il est recommandé d'instancier le widget

directement lorsque ses caractéristiques[1] par défaut vous conviennent et de créer votre widget si vous voulez le modifier. Typiquement on utilise directement la classe bouton mais on hérite de la classe fenêtre, comme on le verra en fin d'article.

2 Programme Minimal

A Code source

```
#include <Gtkmm/main.h>
int main(int argc, char **argv)
{
        Gtk::Main app(argc, argv);
        app.run();
        return 0;
}
```

Le premier objet créé doit être un objet `Gtk::Main`. On lui passe `argc` et `argv` en paramètre. Pour lancer la boucle Gtk on utilise la méthode `run()` de l'instance de `Gtk::Main`.

Ce programme minimal n'a aucun intérêt, il ne fait que lancer la boucle d'évènements sans même donner les moyen de l'arrêter. Il doit donc être arrêté par interruption du processus (signal kill **`Ctrl+C`**). Il montre toutefois la structure générale d'une application construite à l'aide de Gtkmm.

B Compilation & Exécution

Heureusement, Gtkmm utilise `pkg-config` pour nous aider à écrire la ligne de compilation. Vous n'avez donc pas à connaître toutes les librairies nécessaires et les chemins vers ces dernières et les fichiers d'en-têtes. Vous utiliserez toujours au sein de cet article deux lignes de cette forme (en modifiant au besoin les noms des fichiers source et exécutable) :

```
$ g++ `pkg-config Gtkmm-2.4 --cflags --libs` -o Gtkmm_1 Gtkmm_1.cpp
$ ./Gtkmm_1
```

C Une fenêtre

Pour créer un widget de la classe `Window`, on doit instancier un objet de cette dernière. Pour afficher un widget, on utilise la méthode `show()`, comme en Gtk.

```
#include <Gtkmm/main.h>
#include <Gtkmm/window.h>

int main(int argc, char **argv)
{
   Gtk::Main app(argc, argv);
   Gtk::Window w;

   w.show();
   app.run();
   return 0;
```

1 son contenu, son comportement, son apparence, etc.

```
}
```

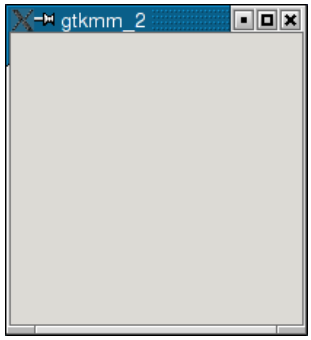
Ce programme crée une fenêtre et l'affiche. Seulement voilà, lorsque l'on ferme la fenêtre, le programme continue de tourner. Cela va surement rappeler aux développeurs C/Gtk+ un mauvais souvenir : le callback nécessaire à toute application pour terminer la boucle Gtk lors de la destruction du widget père (top-level).

En C++/Gtkmm, la vie est bien faite, comme souvent en C++. Il suffit de passer le widget père en paramètre à `app.run()` pour que la boucle Gtk se termine lors de la destruction de ce dernier. En outre, cela rend l'appel à `w.show()` inutile puisque le widget passé en paramètre est affiché par l'objet `Gtk::Main`.

Notre programme afficheur de fenêtre avec fermeture propre devient donc :

```
#include <Gtkmm/main.h>
#include <Gtkmm/window.h>

int main(int argc, char **argv)
{
   Gtk::Main app(argc, argv);
   Gtk::Window w;
   app.run(w);
   return 0;
}
```



3 Application minimale

A Des widgets

Le but de cette application est :

- De présenter une zone de saisie à l'utilisateur.
- De lui offrir un bouton pour qu'il signale qu'il a fini d'utiliser la zone de saisie.
- De lui afficher "Bonjour", suivi de son prénom au dessus de tout cela.

Nous allons, pour ce faire utiliser quatre widgets. Une fenêtre pour afficher l'application, une zone de saisie pour que l'utilisateur saisisse son prénom, un bouton pour informer l'application, qu'il a fini d'entrer son prénom et un champ non éditable (label, ou étiquette) pour afficher le message de bienvenue.

Gtk est un tantinet capricieux, en cela, qu'on ne peux pas prendre une fenêtre et la remplir de widgets... Dans une fenêtre GTK, on met UN widget. Alors, quand on construit une application ayant besoin de plusieurs widgets... Le premier que l'on met dans la fenêtre est un conteneur.

Un conteneur, « container » en anglais, est un widget dont la raison d'être est d'en accueillir d'autres. Il en existe plusieurs types et notamment les conteneurs pouvant accueillir plusieurs widgets. Il existe des conteneurs verticaux, horizontaux, en tableaux et libres. Dans ces derniers, les widgets insérés sont respectivement rangés en colonnes, en lignes, en lignes et colonnes ou enfin librement.

On va utiliser une `Gtk::VBox` (je ne répèterai plus `Gtk::` par la suite) pour cette application, soit une boîte de rangement de widgets verticale.

On ajoutera cette `VBox` à notre `Window` grâce à la méthode `Window::add`.

Une fois le contenant créé, on peut créer le contenu, c'est à dire un `Label`, une `Entry`, et un `Button`. Le constructeur du bouton est le premier constructeur de widget que nous rencontrons prenant un paramètre, il s'agit du texte affiché dans le bouton.

Une fois les widgets créés, on signale à Gtk qu'ils sont prêts en invoquant la méthode `show`.

On ajoute les composants dans la `VBox` créée à cet effet, de haut en bas, à l'aide de la méthode `VBox::pack_start`.

En conséquence de ce que l'on vient de dire, on obtient le code suivant :

```
#include <Gtkmm/main.h>
#include <Gtkmm/window.h>
#include <Gtkmm/box.h>
#include <Gtkmm/label.h>
#include <Gtkmm/entry.h>
#include <Gtkmm/button.h>

int main(int argc, char **argv)
{
   Gtk::Main app(argc, argv);
   Gtk::Window w;
   Gtk::VBox vb;
   w.add(vb);

   Gtk::Label l;
   vb.pack_start(l);
   l.show();
   Gtk::Entry e;
   vb.pack_start(e);
   e.show();
   Gtk::Button b("Bonjour !");
   vb.pack_start(b);
   b.show();

   vb.show();
   app.run(w);
   return 0;
}
```

Rien de très compliqué dans ce code, ce n'est que l'application de ce qui a été dit plus haut. Amusez vous avec l'interface avant de passer à la suite.

B Les signaux

Que rien ne se passe lors d'un clic sur le bouton est relativement frustrant, non ? Nous allons remedier à celà et en profiter pour voir la gestion des signaux en Gtkmm, prise en charge par la librairie `libsigC++`.

Commençons par définir un comportement simple, affichons juste un messages sur la sortie standard.

```
void on_button_clicked() {
	std::cout << "Bonjour" << std::endl;
}
```

Pas de surprise concernant l'affichage du message... Passons à la connexion avec l'action du clic sur le bouton.

On utilise pour ce faire la méthode `connect` des signaux. On connecte un signal non pas à une fonction mais à un slot. Pour transformer un pointeur de fonction en un slot, on utilise la méthode `SigC::slot`. Cette dernière méthode existe en deux formes. Soit elle prend en paramètres un objet et une de ses méthodes, soit une fonction globale. Commençons par voir le cas le plus simple. Transformez le bloc de construction du bouton ci-dessus par :

```
Gtk::Button b("Bonjour !");
b.signal_clicked().connect( sigc::slot(&on_button_clicked) );
vb.pack_start(b);
b.show();
```

On connecte le signal `signal_clicked` du bouton `b` à la fonction globale `on_button_clicked`.

Si vous compilez ce code, vous verrez s'afficher « Bonjour » lorsque vous cliquerez sur le bouton.

C Paramètres de slots

Le but de notre application est d'afficher ce message dans le `Label`, à partir du contenu de l'`Entry`. Le problème est que notre fonction n'a pas accès aux éléments de la fenêtre elle même puisque ces derniers ne sont déclarés que dans la fonction main du programme.

On pourrait résoudre ce problème en déclarant les widgets qui nous intéressent avec une portée globale. Ce ne serait toutefois pas très propre.

On peut également les passer en paramètre au gestionnaire du signal, en utilisant la méthode `SigC::bind`. Cette fonction est en fait un patron prenant en paramètre de patron le type de paramètre que l'on veut passer au slot.

Nous pouvons utiliser cette astuce pour passer UN paramètre mais il nous faut en passer DEUX : l'`Entry` et le `Label`. On crée donc une structure regroupant ces deux entités :

```
struct param {
	Gtk::Entry *e;
	Gtk::Label *l;
};
```

Le bloc de construction du bouton intègre la déclaration du paramètre et la méthode

`SigC::bind` pour devenir :

```
Gtk::Button b("Bonjour !");
param p = { &e, &l };
b.signal_clicked().connect(SigC::bind<param>(SigC::slot(&on_button_clicked), p));
vb.pack_start(b);
b.show();
```

Le gestionnaire de signal peut, dès lors, faire usage des widgets passés en paramètre, au travers de la structure `param p`.

```
void on_button_clicked(param p) {
        p.l->set_text("Bonjour " + p.e->get_text() + " !");
}
```

On voit à présent apparaître dans le gestionnaire deux méthodes extrêmement simples mais très utiles : `Label::set_text` et `Entry::get_text`. Je ne précise pas le rôle de ces méthodes, tant leurs noms sont équivoques et vous invite à consulter la documentation pour découvrir leurs amies.

Il est intéressant de préciser ici que la classe utilisée par Gtkmm pour représenter les chaînes de caractères est `Glib::ustring`, qui est presque équivalente à `std::string`, à cette énorme différence près qu'elle utilise des caractères Unicode et est donc prête pour l'internationalisation de vos programmes.

4 La classe maFenetre

Notre application est d'ores et déjà fonctionnelle. Nous allons toutefois pousser un peu cet article pour aborder une programmation plus propre et logique dans un premier temps puis pour ajouter quelques fioritures à notre interface utilisateur.

La méthode de transmission des widgets au gestionnaire que nous avons utilisée ci dessus n'est pas la plus efficace, ni la plus « propre » à notre disposition. On est ici pour faire du C++, nous allons donc modéliser notre fenêtre avec ses widgets et ses gestionnaires de signaux au moyen d'une classe `maFenetre`.

A Déclaration de la classe

```
class maFenetre : public Gtk::Window
{
protected:
   Gtk::VBox vb;
   Gtk::Label l;
   Gtk::Entry e;
   Gtk::Button b;

   void on_button_clicked();
public:
        maFenetre();
        virtual ~maFenetre();
};
```

On déclare les widgets et gestionnaires de signaux `protected`, de sorte qu'ils puissent être

réutilisés par d'éventuelles classes filles de `maFenetre`. En revanche pour pouvoir construire et détruire la fenêtre, on déclare le constructeur et le destructeur comme publics.

B Définition de la classe

```
maFenetre::maFenetre() : b("Bonjour !") {
   vb.pack_start(l);
   l.show();
   vb.pack_start(e);
   e.show();
   b.signal_clicked().connect(SigC::slot(*this, &maFenetre::on_button_clicked));
   vb.pack_start(b);
   b.show();

   add(vb);
   vb.show();
}

maFenetre::~maFenetre() {
}

void maFenetre::on_button_clicked() {
        l.set_text("Bonjour " + e.get_text() + " !");
}
```

On retrouve dans l'implémentation de la classe `maFenetre` le code précédemment utilisé, juste déplacé, à une exception près. On utilise maintenant une autre forme de `SigC::slot`, prenant en paramètre une classe et un pointeur de fonction. On utilise ici le membre `maFenetre::on_button_clicked` de la classe `maFenetre`. Étant donné que nous sommes dans la portée de la classe `maFenetre`, l'objet passé en paramètre est logiquement `*this`.

N'oubliez surtout pas de spécifier entièrement le gestionnaire de signal, sous peine d'être gratifié d'injures par g++ semblables à :

```
$ g++ `pkg-config Gtkmm-2.0 --cflags --libs` -o Gtkmm_6 Gtkmm_6.cpp
Gtkmm_6.cpp: Dans constructor « maFenetre::maFenetre() »:
Gtkmm_6.cpp:28: error: le C++ ISO interdit de prendre l'adress d'un membre de
  fonction non statique non qualifié pour former un pointeur d'un membre de
  fonction. Disons «&maFenetre::on_button_clicked»
```

Cette erreur provient du fait qu'on ait écris :

```
b.signal_clicked().connect(SigC::slot(*this, &on_button_clicked));
```

C Boucle principale

```
int main(int argc, char **argv)
{
   Gtk::Main app(argc, argv);
   maFenetre w;

   app.run(w);
   return 0;
}
```

Maintenant que nous avons fini par écrire le `main`, force est de constater que le code ainsi

organisé est plus clair et plus logique. L'objet graphique que vous avez devant vos yeux ébahis par tant de beauté lors de l'exécution correspond à un objet logique dans votre programme. Ce dernier possède les objets qui le constituent et les gestionnaires de signaux qu'il est susceptible d'intercepter (et pouvant nous intéresser, il intercepte par ailleurs une foule de trucs inutiles ...).

D Décoration

Pour finir, nous allons utiliser quelques méthodes pratiques de la classe `Gtk::Window` permettant d'affecter un icône à notre application ou de lui donner un titre plus sympa que le nom de l'exécutable lancé. Il existe bien entendu d'autres méthodes utilitaires que je vous laisse le soin de découvrir par vous mêmes.

`set_title` donne un titre à la fenêtre, son paramètre est une chaîne de caractères.

`set_icon` affecte un icône à la fenêtre. Selon les gestionnaires de fenêtres, ce dernier est affiché dans la barre d'applications (barre des tâches) et/ou dans la barre de titre de l'application. C'est également lui qui est utilisé lorsque vous minimiser votre fenêtre.

Cette méthode prend en paramètre un `const Glib::RefPtr<Gdk::Pixbuf>&`... Ce qui explique qu'ici, on préfère utilise plutôt une version modifiée, `set_icon_from_file` qui prend comme paramètre, de type `const std::string&`, le chemin vers un fichier image.

Le code de l'application terminée (en un seul fichier pour plus de lisibilité) :

```
#include <Gtkmm/main.h>
#include <Gtkmm/window.h>
#include <Gtkmm/box.h>
#include <Gtkmm/label.h>
#include <Gtkmm/entry.h>
#include <Gtkmm/button.h>

class maFenetre : public Gtk::Window
{
protected:
   Gtk::VBox vb;
   Gtk::Label l;
   Gtk::Entry e;
   Gtk::Button b;

   void on_button_clicked();
public:
        maFenetre();
        virtual ~maFenetre();
};

maFenetre::maFenetre() : b("Bonjour !") {
   set_title("Bonjour toi !");
   set_icon_from_file("hi.png");
   vb.pack_start(l);
   l.show();
   vb.pack_start(e);
   e.show();
   b.signal_clicked().connect(SigC::slot(*this, &maFenetre::on_button_clicked));
   vb.pack_start(b);
   b.show();

   add(vb);
   vb.show();
}
```

```
maFenetre::~maFenetre() {
}

void maFenetre::on_button_clicked() {
        l.set_text("Bonjour " + e.get_text() + " !");
}

int main(int argc, char **argv) {
   Gtk::Main app(argc, argv);
   maFenetre w;

   app.run(w);
   return 0;
}
```

5 Conclusion

Il vous a été sommairement présenté Gtkmm. Il est important de signaler que l'on n'a utilisé ici que les bases de ce que nous pouvons faire avec Gtkmm, pour donner une idée des fonctionnalités uniquement. Il vous reste à creuser le sujet par vous mêmes en utilisant la documentation présente sur Internet.

6 Liens & Références

[1] http://www.xgarreau.org/aide/devel/langtk/cpp_gtkmm.php

[2] glibmm, Gtkmm & gnomemm : http://gtkmm.sourceforge.net

[3] La bible Gtkmm et associés : http://gtkmm.sourceforge.net/Gtkmm2/docs/

[4] libsigC++ : http://libsigc.sourceforge.net

[5] bakery : http://bakery.sourceforge.net