PoP Série 4 niveau 0

Usage de GTKmm 4 pour l'interface graphique utilisateur : Création d'une fenêtre et dessin avec Cairo

manuel de référence en-ligne de GTKmm 4 (version fr)

Exercice 1.(niveau 0) : **Création de votre première fenêtre avec GTKmm 4**

Dans le fichier archive **PoP_s4_niv0_code_GTKmm4**, vous trouverez plusieurs fichiers utiles pour ce premier contact avec GTKmm 4. Les fichiers ont été compilés sur la VM avec les commandes et le Makefile fournis. Récupérez tout d'abord le fichier **simple.cc** du répertoire **GTKsimple**.

Cet exemple va créer une fenêtre vide de taille 500x500 pixels avec le texte «Série4 niveau 0» visible sur le bandeau supérieur. Il est possible d'avoir un warning dans le terminal mais cela n'empêche pas la création de la fenêtre quelque part sur votre écran.

```
#include <gtkmm.h>

class MyWindow : public Gtk::Window
{
public:
    MyWindow();
};

MyWindow::MyWindow()
{
    set_title("Série4 niveau 0");
    set_default_size(500, 500);
}

int main(int argc, char* argv[])
{
    auto   app = Gtk::Application::create("org.gtkmm.examples.base");
    return app->make_window_and_run<MyWindow>(argc, argv);
}
```

Au tout début la directive inclut l'interface (ou API) de GTKmm avec la syntaxe <gtkmm.h>. Par la suite on précisera plutôt quelle fonctionnalité de gtkmm on désire plutôt que de tout inclure car ce fichier en-tête est énorme.

Ensuite chaque fois que nous voudrons appeler une fonction/méthode ou déclarer un object de GTK, il faudra le faire précéder de **Gtk::** pour indiquer son appartenance à GTKmm. C'est une excellente pratique qui permet de documenter le code car on connait mieux la nature des objets et des méthodes.

Ensuite remarquez la forme étendue de **main()** avec les deux paramètres **argc** et **argv** déjà abordée au semestre d'automne (Topic 11); c'est nécessaire car ces 2 paramètres sont immédiatement transmis à la méthode **make_window_and_run**.

Mais tout d'abord la fonction **Gtk ::Application ::create** permet de créer le pointeur **app** sur une application Gtk. Ce nouvel objet va gérer tout ce qui concerne notre première application

GTKmm ; il faudra toujours commencer par en créer un. Le pointeur **app** est immédiatement utilisé pour appeler la méthode **make_window_and_run** qui comme son nom l'indique crée d'abord une instance de fenêtre de type **MyWindow** qui hérite de la classe parente **Gtk ::Window**. Comme nous avons le contrôle sur cette classe dérivée, nous pouvons choisir le texte qui apparait dans son bandeau de titre en appelant la méthode **set_title()** dans son constructeur par défaut. Sa taille est aussi choisie dans le constructeur.

La seconde action de la méthode **make_window_and_run** d'appeler une méthode qui s'appelle **run** qui lance une boucle infinie responsable de traiter les *événements d'interaction* au fur et à mesure de leur création. Par exemple vous pouvez déplacer la fenêtre ou changer sa taille : on appelle chacune de ces actions un *événement*. Chaque événement est traité par GTKmm sous forme d'une réaction prédéfinie, par exemple redessiner le contenu de la fenêtre. Par la suite vous allez pouvoir définir vous-même la nature des réactions attendues pour les événements qui vous intéressent, par exemple quand on clique sur un bouton.

c) Voici la syntaxe de la commande **Build** de geany (l'ordre est très important) :

```
g++ -std=c++17 -Wall  -o "%e" "%f" `pkg-config gtkmm-4.0 --cflags --libs`
```

d) Vous pouvez ré-utiliser/étendre ce Makefile pour les futurs programmes avec GTKmm 4

```
#
# GTKmm4  avec -std=c++17
#

OUT = test
CXX = g++
CXXFLAGS = -Wall -std=c++17
LINKING = `pkg-config --cflags gtkmm-4.0`
LDLIBS = `pkg-config --libs gtkmm-4.0`
OFILES = simple.o

all: $(OUT)

simple.o: simple.cc
	$(CXX) $(CXXFLAGS) $(LINKING) -c $< -o $@ $(LINKING)

$(OUT): $(OFILES)
	$(CXX) $(CXXFLAGS) $(LINKING) $(OFILES) -o $@ $(LDLIBS)

clean:
	@echo "Cleaning compilation files"
	@rm *.o $(OUT) *.cc~ *.h~
```

Activité : récupérer le code source de simple.cc, le compiler avec geany ou avec la commande **make** puis lancer l'exécutable.

Examiner les actions que vous pouvez faire sur la fenêtre ainsi créée. Editer le texte du bandeau et changez sa taille par défaut. Pour quitter le programme il suffit de fermer la fenêtre ou de faire Ctrl-C dans le terminal de lancement de l'exécutable.

Exercice 2.(niveau 0) : **Création de votre première application interactive avec GTKmm 4**

Cet exemple illustre les notions *d'événement*, de *signal* et de *signal handler*[1] produisant une réaction souhaitée quand on clique sur un bouton. Etant composée de deux modules un fichier Makefile est fourni.

A la différence du précédent exercice, ici vous allez découvrir une application dont la réaction (=> *signal handler*) peut être est définie par vous-même. Le module principal présente une petite différence avec l'exemple précédent : cette différence est un nouveau nom de type entre les chevrons : **<HelloWorld>**. Ce type de construction du C++ est appelé un *template* ; c'est comme si on passait un nom de type en paramètre et cela permet de ré-utiliser le même code pour différents types. Ce sujet dépasse le programme du cours qui ne traitera pas la conception de *template* ; nous aurons seulement à utiliser cette syntaxe dans le cadre de GTKmm4.

```
#include "helloworld.h"
#include <gtkmm/application.h>

int main (int argc, char *argv[])
{
    auto app = Gtk::Application::create("org.gtkmm.example");

    //Shows the window and returns when it is closed.
    return app->make_window_and_run<HelloWorld>(argc,argv);
}
```

Examinons les éléments importants :

Au tout début une directive inclut l'interface du module helloworld dédié à la classe **Helloworld**. La directive d'inclusion de gtkmm est plus limitée que la précédente car on n'a pas besoin de tout gtkmm. On retrouve la création de l'application comme pour l'exercice 1. La nouveauté vient de l'appel de la méthode **make_window_and_run<HelloWorld>.** En examinant l'interface de la classe Helloworld on voit que le type HelloWorld est aussi **une classe dérivée de la classe Gtk ::Window.**

```
#ifndef GTKMM_EXAMPLE_HELLOWORLD_H
#define GTKMM_EXAMPLE_HELLOWORLD_H

#include <gtkmm/button.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window  // heritage
{
public:
    HelloWorld();
    ~HelloWorld() override;  // sera vu avec le MOOC Polymorphisme

protected:
    //Signal handlers:
    void on_button_clicked();

    //Member widgets:
    Gtk::Button m_button;
};
#endif // GTKMM_EXAMPLE_HELLOWORLD_H
```

[1] On trouve aussi l'expression ***fonction callback*** pour illustrer l'idée d'un appel d'une fonction en réaction à quelque chose. Cependant elle est moins utilisée actuellement.

En plus d'un constructeur par défaut et d'un destructeur suivi de **override** (explications la semaine prochaine) on trouve deux éléments **protected** :

- la méthode **on_button_clicked()** ⇔ *signal handler*
- l'attribut **Button m_button** qui est un widget qui va exister dans la fenêtre de l'application Helloworld

Examinons l'implémentation de la classe **Helloworld**. L'essentiel se passe dans le constructeur de la classe. Celui-ci initialise le widget **m_button** avec un nom dans la liste d'initialisation.

L'élément le plus important est la création du lien entre :

1. Le widget **m_button**
2. Et le signal handler **on_button_clicked**
3. Cette création précise que la méthode 2) réagit au signal **signal_clicked**
4. Le lien est créé avec la méthode **connect()** qui reçoit en paramètre un pointeur de fonction sur le signal handler

Sans entrer dans les détails, c'est grâce à la mémorisation du pointeur de fonction que ladite fonction pourra être appelée automatiquement plus tard à chaque fois que le signal **signal_clicked** est activé quand l'utilisateur produit *l'événement* de cliquer sur le bouton **m_button** dans la fenêtre **Helloworld**.

```
#include "helloworld.h"
#include <iostream>

HelloWorld::HelloWorld()
: m_button("Hello World")   // creates a new button with label "Hello World".
{
    // Sets the margin around the button.
    m_button.set_margin(10);

    // When the button receives the "clicked" signal, it will call the
    // on_button_clicked() method defined below.
    m_button.signal_clicked().connect(sigc::mem_fun(*this,
                                                    &HelloWorld::on_button_clicked));

    // This packs the button into the Window (a container).
    set_child(m_button);
}

HelloWorld::~HelloWorld()
{
}

void HelloWorld::on_button_clicked()
{
    static unsigned count(0);
    std::cout << ++count << "Hello World \a" << std::endl;
}
```

Le widget est ajouté à la fenêtre Helloworld avec la méthode **set_child.**

Dans la version que nous vous fournissons, l'action du signal handler **on_button_clicked** est complétée par l'affichage d'un compteur qui mémorise le nombre de fois qu'il a été appelé ainsi que par l'affichage du caractère **\a** qui devrait produire un « bip ».

Activité : récupérer le code source du module Helloworld et main.cc, le compiler avec la commande **make** puis lancer l'exécutable.

Examiner les actions que vous pouvez faire sur la fenêtre ainsi créée (quel est le texte sur le bandeau de la fenêtre ? sa taille ? bordure cosmétique ?). Pour voir l'action du bouton il faut lancer l'exécutable depuis un terminal.

Un petit pas pour le projet : modifier le bouton pour qu'il permette de quitter le programme ; c'est-à-dire changer le texte fourni à l'initialisation par un nouveau texte « Quit » pour être conforme à cette nouvelle fonction. Ensuite modifier le signal handler pour appeler la fonction exit au lieu de faire un affichage.

Exercice 3.(niveau 0) : **Création de votre premier dessin avec GTKmm 4**

Pour cet exemple, le module principal reprend le même principe que l'exemple précédent :

```
#include "myarea.h"
#include <gtkmm/application.h>
#include <gtkmm/window.h>

class ExampleWindow : public Gtk::Window
{
public:
    ExampleWindow();

protected:
    MyArea m_area;
};

ExampleWindow::ExampleWindow()
{
    set_title("DrawingArea");
    set_child(m_area);
}

int main(int argc, char** argv)
{
   auto app = Gtk::Application::create("org.gtkmm.example");

   return app->make_window_and_run<ExampleWindow>(argc,argv);
}
```

On retrouve la création de l'application comme pour les exercices 1 et 2. Cette fois ci avec le type **<ExampleWindow>**. La nouveauté vient de la présence d'un attribut **m_area** de la classe **MyArea** dans la classe ExampleWindow.

Cette classe **MyArea** est elle-même dérivée de la classe **DrawingArea** comme on peut le voir dans la déclaration de la classe dans **myarea.h**. Mais tout d'abord deux mots sur le type **DrawingArea** qui est spécialisé pour le dessin. Il est important de souligner que la convention d'axe retenue par GTKmm est celle des systèmes gestionnaires de fenêtres pour lesquels :

- l'**origine est en haut à gauche** de la fenêtre,
- l'axe X croît vers la droite et varie entre 0 et **width** (largeur de la fenêtre en **pixels**)
- l'axe Y croît vers le bas et varie entre 0 et **height** (hauteur de la fenêtre en **pixels**)

Le Modèle doit rester indépendant de la Visualisation : de son coté, le Modèle doit aussi avoir défini son système d'axes qui dépend du problème qu'il doit résoudre en termes d'unité, d'échelle, d'espace couvert et d'orientation des axes principaux (on adopte les conventions mathématiques classiques d'axe Y croissant positivement vers le haut). C'est généralement une faiblesse de conception de s'aligner sur la convention de visualisation car celle-ci peut changer d'une bibliothèque à une autre ; par exemple OPEN-GL n'a pas la même convention d'axes que GTKmm. C'est la responsabilité du système de visualisation de mettre en place les formules de changement de système de coordonnées entre ces deux conventions (cf cours et série ultérieure). Dans cet exemple le dessin est effectué dans l'espace de la fenêtre GTKmm.

L'interface du module **myarea.h** nous donne des précisions sur la méthode de dessin. On y voit sous **protected** une méthode **on_draw** qui est responsable de faire le dessin dès que le widget reçoit le signal de rafraichir l'affichage *car* un pointeur de fonction vers **on_draw** est mémorisé avec l'appel **set_draw_func** dans le constructeur de **MyArea** de myarea.cc.

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>

class MyArea : public Gtk::DrawingArea  // héritage
{
public:
    MyArea();
    virtual ~MyArea();

protected:
    void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
                 int width,int height);
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

L'autre nouveauté de cette déclaration est le paramètre **cr** de **on_draw**. Ce paramètre **cr** est (en gros) une référence constante sur un objet **Cairo::Context** qui permet de *mémoriser tous les paramètres courants du dessin*. Il serait en effet fastidieux de fournir un à un chacun des paramètres de dessin à **on_draw** ou aux méthodes de dessin spécialisées. Un *contexte* sert donc à mémoriser l'état courant de paramètres tels que l'épaisseur ou la couleur du trait.

Voyons maintenant les commandes de dessin utilisées dans l'implémentation de notre méthode **on_draw**. On dispose en paramètre de la largeur (**width**) et de la hauteur (**height**) de la fenêtre. Rappel : le dessin s'effectue dans l'espace de la fenêtre **[0, width]** x **[0, height].** On se sert d'ailleurs de ces valeurs pour déterminer le centre de la fenêtre **(xc, yc)**.

Tout d'abord, on spécifie une couleur noire avec **set_source_rgb** qui est utilisée par **paint()** pour mettre à jour la couleur du fond. Après le calcul du centre, on appelle des méthodes à l'aide du pointeur sur l'objet **Context** obtenu en paramètre. On distingue les méthodes :

- qui permettent de changer l'état d'un paramètre du dessin
    - **set_line_widt**h(val en pixel)
    - **set_source_rgb(r,g,b)** avec chacune des valeurs dans [0,1]

- qui permettent de créer le dessin
    - **move_to(x, y)** : définit un début de ligne pour un *path*
    - **line_to(x ,y)** : définit un tracé entre la précédente valeur fournie et (x,y) dans le path courant.
    - **stroke()** : dessine le tracé du *path* qui vient d'être construit avec **move_to** et **line_to** puis détruit ce *path*.

```
#include "myarea.h"
#include <cairomm/context.h>

MyArea::MyArea()
{
  set_draw_func(sigc::mem_fun(*this, &MyArea::on_draw));
}

MyArea::~MyArea(){}

void MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr
                     int width, int height)
{
  // changing the background color to black
  cr->set_source_rgb(0., 0., 0.);
  cr->paint();

  // center of the window
  int xc(width/2), yc(height/2);

  cr->set_line_width(10.0);

  // draw red lines out from the center of the window
  cr->set_source_rgb(0.8, 0.0, 0.0);
  cr->move_to(0, 0);
  cr->line_to(xc, yc);
  cr->line_to(0, height);
  cr->move_to(xc, yc);
  cr->line_to(width, yc);
  cr->stroke();
}
```

Activité :

- Editer la forme qui est dessinée en ajoutant/enlevant des appels **move_to** et **line_to**.
- Intercaler des commandes de changement de couleur et de largeur du trait
- Comment le dessin change-t-il lorsqu'on change la taille de la fenêtre ?