PoP C++ Série 5 niveau 0

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

Traduit du manuel de référence en-ligne de GTKmm 3

Exercice 1.(niveau 0) : **Création de votre première fenêtre avec GTKmm 3**

a) Pour ceux qui travaillent avec la VM sur leur laptop, la VM du second semestre, disponible depuis le 6 mars, est nécessaire pour faire cette série et la suite du projet.

b) Dans le fichier archive **serie5_PoP_code_source**, vous trouverez plusieurs fichiers utiles pour ce premier contact avec GTKmm 3. Les fichiers ont été compilés sur la VM avec les commandes et le Makefile fournis. Récupérez tout d'abord le fichier **simple.cc**

```
#include <gtkmm.h>

using namespace std;

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

	Gtk::Window window;
	window.set_default_size(200,200);

	return app->run(window);
}
```

Au tout début la directive inclut l'interface (ou API) de GTKmm avec la syntaxe <gtkmm.h>

Ensuite il n'y a que l'instruction **using** pour la bibliothèque standard, PAS pour GTKmm. Cela veut dire que 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** ; c'est nécessaire car ces 2 paramètres sont immédiatement transmis à la méthode de classe **Gtk ::Application :: create**().

La fonction **create** permet de créer un objet de type **Gtk ::Application** dont on récupère un pointeur (de type *smart pointer* qui est plus robuste que les pointeurs à-la-C). Ce nouvel objet va gérer tout ce qui concerne notre première application GTKmm ; il faudra toujours commencer par en créer un.

Un objet **Gtk ::Window** est créé pour que notre application dispose d'une fenêtre. La méthode s**et-default_size(200,200)** définit la taille qu'elle aura à l'initialisation. Par la suite l'utilisateur peut interactivement changer la taille de cette fenêtre.

On associe cette fenêtre **window** à l'application en la passant en paramètre à la méthode **run** de l'objet **app** ; comme c'est un pointeur on doit faire cet appel avec l'opérateur -> .

C'est en fait dans ce dernier appel que l'exécution du programme va passer le plus de temps car la méthode **run** lance une boucle infinie qui traite 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++11 -Wall  -o "%e" "%f" `pkg-config gtkmm-3.0 --cflags --libs`
```

d) Vous pouvez ré-utiliser/étendre ce Makefile pour les futurs programmes avec GTKmm 3

```
OUT = test
CXX = g++
CXXFLAGS = -Wall -std=c++11
LINKING = `pkg-config --cflags gtkmm-3.0`
LDLIBS = `pkg-config --libs gtkmm-3.0`
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. 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 3**

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 helloworld_Makefile est fourni qu'il faut renommer en Makefile pour travailler avec make plutôt qu'avec geany.

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 peu de différences avec l'exemple précédent.

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

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

  HelloWorld helloworld;

  //Shows the window and returns when it is closed.
  return app->run(helloworld);
}
```

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 la création d'un objet de la classe **Helloword**. C'est cet objet qui est transmis à la méthode **run** de l'application plutôt qu'un objet de type **Window**. En examinant l'interface de la classe Helloworld on voit qu'il n'y a pas d'incohérence dans l'appel de la méthode **run** car **: la classe Helloworld est une sous-classe 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();
  virtual ~HelloWorld();  // virtual est hérité de la classe parente Window

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 utilisé actuellement.

En plus d'un constructeur par défaut et d'un destructeur précédé de **virtual** (c'est nécéssaire pour des questions de gestion de mémoire allouée dynamiquement) 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 border width of the window. // cosmetique
  set_border_width(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).
  add(m_button);

  // The final step is to display this newly created widget...
  m_button.show();
}

HelloWorld::~HelloWorld()
{
}

void HelloWorld::on_button_clicked()
{
  std::cout << "Hello World" << std::endl;
}
```

Le widget est ajouté à la fenêtre Helloworld avec la méthode **add** puis affiché avec **show**.

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 (taille ? bordure cosmetique ?). 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 son nom à l'initialisation 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 3**

Nous exploitons ici le widget **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). Des méthodes sont disponibles pour faciliter cette étape.

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>

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

   Gtk::Window win;
   win.set_title("DrawingArea");

   MyArea area;
   win.add(area);
   area.show();

   return app->run(win);
}
```

On retrouve la création de l'application comme pour les exercices 1 et 2. Cette fois ci on crée une fenêtre **win** comme dans l'exercice 1. La nouveauté vient de la création d'un objet **area** de la classe **MyArea**. Cette classe **MyArea** est en fait dérivé de la classe **DrawingArea** comme on peut le voir dans la déclaration de la classe dans myarea.h.

Il se trouve que **Drawing Area** est une sorte de widget tout comme l'objet Button de l'exemple 2. Et, tout comme le widget m_button de l'exemple2, on peut l'ajouter le nouvel objet **area** à la fenêtre **win** ; c'est ce qui est fait immédiatement après sa déclaration dans ce programme. Ensuite on lance l'application comme d'habitude en lui passant la fenêtre **win**.

L'interface du module myarea.h nous donne des précisions sur la méthode de dessin. On y voit sous protected une méthode hérité **on_draw** qui est responsable de faire le dessin dès que le widget reçoit le signal de rafraichir l'affichage.

Contrairement à ce que nous avons vu jusqu'à maintenant il ne s'agit pas d'une surcharge de la méthode de la classe parente mais d'une *redéfinition complète* de la méthode hérité. Cette redéfinition est nécessaire car le prototype de **on_draw** ne change PAS ; on veut travailler avec le même prototype que la classe parente mais on veut que ça soit notre version de la classe MyArea qui soit exécutée. Donc contrairement à la surcharge qui définit une *variante*, ici c'est un *remplacement*. On le signale au compilateur avec le mot clef **override** à la suite du prototype.

```
#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:
  //Override default signal handler:
  bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override;
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

L'autre nouveauté de cette déclaration est justement l'unique paramètre 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 y trouve au tout début la déclaration d'un objet **Allocation** qui sert à mémoriser l'emplacement et la taille d'un widget ; dans not cas on s'intéresse seulement à récupérer sa taille en pixel selon la largeur (**width**) et la hauteur (**height**) car 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)**.

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

MyArea::MyArea(){}

MyArea::~MyArea(){}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();

  // coordinates for the center of the window
  int xc, yc;
  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();

  return true;
}
```

Après le calcul du centre, on appelle des méthodes à l'aide du pointeur sur l'objet **Context** obtenu en paramètre. On distingue deux types de méthodes :

- celles permettant 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]

- celles qui contribuent au 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*.

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 ?