PoP C++ Série 6 niveau 0

Usage de GTKmm pour l'interface graphique utilisateur : Création d'une petite interface

Adapté du manuel de référence en-ligne de GTKmm 3

Exercice 1.(niveau 0) : **Layout et gestion d'evenement**

Dans cet exemple nous allons avoir une interaction entre les boutons et l'affichage.

1.1) Le programme principal définit un widget de la classe que nous avons définie. La taille par défaut **330x200** est celle que prendrait la fenêtre si on n'empêchait pas de lui changer sa taille avec la ligne suivante qui met l'attribut **resizable** à faux. Le comportement du programme va dépendre également du contrôle de taille que nous allons demander sur le widget MyArea.

```
#include "myevent.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");

   MyEvent eventWindow;
   eventWindow.set_default_size(300, 200);
   eventWindow.set_resizable(false);

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

Activité : commenter successivement les lignes fixant la taille par défaut et modifiant l'attribut resizable et constater le comportement à l'exécution.

1.2) Passons à l'interface du module **myevent** :

Il contient l'interface de deux classes :

- La classe **MyArea** que nous avons étendue avec 2 méthodes publiques **clear**() et **draw**(), une méthode privée **refresh()** et un attribut privé (booléen **empty**).

- La classe Myevent conient 2 attributs Button avec leur signal_handler. Elle possède aussi un attribut **MyArea** et plusieurs **Gt ::Box** pour le layout. Enfin on remarque une fonction privée **draw**.

```
#ifndef GTKMM_EXAMPLE_MYEVENT_H
#define GTKMM_EXAMPLE_MYEVENT_H

#include <gtkmm.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();
  void clear();
  void draw();

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

private:
  bool empty;
  void refresh();
};


class MyEvent : public Gtk::Window
{
public:
  MyEvent();
  virtual ~MyEvent();

protected:
  //Button Signal handlers:
  void on_button_clicked_clear();
  void on_button_clicked_draw();

  Gtk::Box m_Box, m_Box_Top, m_Box_Bottom;
  MyArea          m_Area;
  Gtk::Button     m_Button_Clear;
  Gtk::Button     m_Button_Draw;

private:
      void draw();
};

#endif // GTKMM_EXAMPLE_MYEVENT_H
```

L'implémentation définit d'abord MyArea.

- Les méthodes publiques **clear**() et **draw**() sont utilisées pour le contrôle de l'interaction avec les boutons. Elles agissent sur le booléen **empty :**
    - La méthode **clear** met le booleen **empty** à vrai et appelle **refresh**
    - La méthode **draw** met le boolen **empty** à faux et appelle **refresh**

- La méthode **refresh** produit un *signal* qui va causer un appel du signal handler **on_draw**

```
#include <iostream>
#include "myevent.h"
#include <cairomm/context.h>

using namespace std;

MyArea::MyArea(): empty(false)
{
}

MyArea::~MyArea()
{
}

void MyArea::clear()
{
  empty = true;
  refresh();
}

void MyArea::draw()
{
  empty = false;
  refresh();
}

void MyArea::refresh()
{
  auto win = get_window();
  if(win)
  {
        Gdk::Rectangle r(0,0, get_allocation().get_width(),
                              get_allocation().get_height());

        win->invalidate_rect(r,false);
  }
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  if(not empty)
  {
        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();
  }
  else
  {
        cout << "Empty !" << endl;
  }

  return true;
}
```

```
//-------------------------------------

MyEvent::MyEvent() :
  m_Box(Gtk::ORIENTATION_VERTICAL,10),
  m_Box_Top(Gtk::ORIENTATION_HORIZONTAL, 10),
  m_Box_Bottom(Gtk::ORIENTATION_HORIZONTAL, 10),
  m_Button_Clear("Clear"),
  m_Button_Draw("Draw")
{
  // Set title and border of the window
  set_title("Drawing Area and Buttons");
  set_border_width(0);

  // Add outer box to the window (because the window
  // can only contain a single widget)

  add(m_Box);

  //Fill the outer box:
  m_Box.pack_start(m_Box_Top);
  m_Box.pack_start(m_Box_Bottom);

  m_Area.set_size_request(200,200);
  m_Box_Top.pack_start(m_Area);

  m_Box_Bottom.pack_start(m_Button_Clear,false,false);// keep fixed width
  m_Box_Bottom.pack_start(m_Button_Draw,false,false); // and aligned to left;

  // Connect the clicked signal of the button to
  // their signal handler
  m_Button_Clear.signal_clicked().connect(sigc::mem_fun(*this,
              &MyEvent::on_button_clicked_clear) );

  m_Button_Draw.signal_clicked().connect(sigc::mem_fun(*this,
              &MyEvent::on_button_clicked_draw) );

  // Show all children of the window
  show_all_children();
}

MyEvent::~MyEvent()
{
}

void MyEvent::on_button_clicked_clear()
{
  cout << "Clear" << endl;
  m_Area.clear();
}

void MyEvent::on_button_clicked_draw()
{
  cout << "Draw" << endl;
  m_Area.draw();
}
```

L'aspect intéressant ci est l'uage des Gtk ::Box pour définir l'empalcement relatif des widgets. On utilise des **Gtk ::Box** qui sont initialisés dans la liste d'initialisation en précisant si leur contenu doit s'aligner verticalement (**`Gtk::ORIENTATION_VERTICAL`**) ou horizontalement (**`Gtk::ORIENTATION_HORIZONTAL`**). On remarque également qu'on peut fixer la taille du widget MyArea avec un appel de **set_size_request**(200,200).

Activité : examiner comment les conteneur Gtk ::Box sont utilisés et effectuez des modifications, par exemple changer le remplissage de Vertical à horizontal et vice-versa.