PoP Série 6 niveau 0

Usage de GTKmm 4:
Fenêtre de dialogue / distorsion / event du clavier / Timer

Code source adapté du manuel de référence en-ligne de GTKmm 4

Exercice 1.(niveau 0) : **usage d'événements provenant des touches du clavier**

Cet exercice introduit la gestion d'événements provenant du clavier montrant également la possibilité de changement du texte d'un bouton (code exécuté en cours en semaine6). Vous trouverez les détails des explications sur la gestion d'événements provenant des Buttons dans la série5 niveau0 ; le module main.cc étant le même on ne le détaille pas ici. On se concentre ici sur les quelques différences introduites par rapport à cet exercice de référence.

La nouveauté dans l'interface du module **myevents** est la méthode du handler des événements du clavier :

bool on_window_key_pressed(guint keyval, guint keycode, Gdk::ModifierType state);

Dans myevent.cc sa définition se concentre sur l'utilisation du premier paramètre **`guint keyval`** qui est converti vers le standard UNICODE avec la méthode **`gdk_keyval_to_unicode`**.

On peut alors utiliser un simple switch avec autant de case que de touches clavier qui nous intéressent. Nous avons retenu quelques utilisations :

- '**w**' fait afficher un message dans le terminal
- '**c**' et '**C**' modifient le texte affiché par un bouton avec la méthode **set.label**(…).
- '**q'** quitte le programme en appelant **hide()** ou **exit**(0).

Activité : ajouter d'autres réactions liées à d'autres touches du clavier.

Exercice 2 (niveau 0) : **contrôle du temps avec les événements d'un Timer**

Cet exercice simplifie autant que possible l'exemple Timerexample fourni par le manuel GTKmm 4 dans le sens où on gère un seul Timer au lieu d'un nombre quelconque. Le programme principal définit un widget BasicTimer dérivé de la classe Window

```
#include "basictimer.h"
#include <gtkmm/application.h>
int main (int argc, char *argv[])
{
    auto app = Gtk::Application::create();
    return app->make_window_and_run<BasicTimer>(argc, argv);
}
```

Le but est de faire exécuter une action particulière à intervalle régulier, dans cet exemple toutes les 0,5 s. L'exemple illustre également comment arrêter puis relancer le Timer ainsi que comment faire afficher du texte permanent ou la valeur d'une variable avec **Gtk ::Label**.

```
#ifndef GTKMM_EXAMPLE_TIMEREXAMPLE_H
#define GTKMM_EXAMPLE_TIMEREXAMPLE_H

#include <gtkmm.h>
#include <iostream>

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

protected:
  // button signal handlers
  void on_button_add_timer();
  void on_button_delete_timer();
  void on_button_quit();

  // This is the standard prototype of the Timer callback function
  bool on_timeout();

  // Member data:
  Gtk::Box    m_Box, top_box, bottom_box;
  Gtk::Button m_ButtonAddTimer, m_ButtonDeleteTimer, m_ButtonQuit;
  Gtk::Button start_timer, stop_timer, quit_button;
  Gtk::Label  text_label, data_label;

  // Keep track of the timer status (created or not)
  bool timer_added;

  // to store a timer disconnect request
  bool disconnect;
  // This constant is initialized in the constructor's member initializer:
  const int timeout_value;
};

#endif // GTKMM_EXAMPLE_TIMEREXAMPLE_H
```

<u>Interface du module</u> :

Les actions de lancer puis d'arrêter le Timer sont demandées à l'aide des Buttons **start_timer** et **stop_timer**. Chacun dispose de son signal handler qui va respectivement contrôler ces actions de lancer ou arrêter le Timer.

La Box **m_Box** sert à disposer les Buttons horizontalement dans la Box top_box, puis le texte avec des Label dans la Box bottom_box.

Le booléen **timer_added** sert à mémoriser si un Timer a déjà été créé pour éviter des actions incorrectes (ex : créer plusieurs Timers ou détruire un Timer qui n'existe pas).

Le booléen **disconnect** va servir de relai pour une demande d'arrêt du Timer.

La *constante* entière **timeout_value** est un nombre de millisecondes qui sera initialisée à la déclaration de l'instance de SimpleTimerExample.

Implémentation du module :

Le constructeur précise les label des Buttons dans la liste d'initialisation (Start, Stop et Quit). On y trouve aussi la valeur de la période du Timer, ici 500ms ; vous pourrez la modifier selon vos besoins. Enfin the Timer n'existe pas au lancement du programme ; le booléen **timer_added** est initialisé à **false** ainsi que le booléen **disconnect**.

Le constructeur dispose les Buttons dans la Box horizontalement (conformément au paramètre de la liste d'initialisation) et les connectent à leur signal handler. L'appel de la méthode **expand()** autorise les boutons à changer de taille quand la fenêtre change de taille.

```
#include "basictimer.h"

BasicTimer::BasicTimer() :
      m_box(Gtk::Orientation::VERTICAL, 10),
      top_box(Gtk::Orientation::HORIZONTAL, 10),
      bottom_box(Gtk::Orientation::HORIZONTAL, 10),
      start_timer("_Start", true),
      stop_timer("_Stop", true),
      quit_button("_Quit", true),
      text_label("Simulation step : "),
      data_label("0"),
      timer_added(false),// to handle a single timer
      disconnect(false), // to handle a single timer
      timeout_value(500) // 500 ms = 0.5 seconds
{
      m_box.set_margin(10);
      set_child(m_box);

      m_box.append(top_box);
      m_box.append(bottom_box);

      top_box.append(start_timer);
      top_box.append(stop_timer);
      top_box.append(quit_button);

      start_timer.set_expand();
      stop_timer.set_expand();
      quit_button.set_expand();

      bottom_box.append(text_label);
      bottom_box.append(data_label);

      text_label.set_expand();
      data_label.set_expand();

      // Connect the three buttons:
      quit_button.signal_clicked().connect(sigc::mem_fun(*this,
                  &BasicTimer::on_button_quit));
      start_timer.signal_clicked().connect(sigc::mem_fun(*this,
                  &BasicTimer::on_button_add_timer));
      stop_timer.signal_clicked().connect(sigc::mem_fun(*this,
                  &BasicTimer::on_button_delete_timer));
}
```

Suite page suivante…

```
void BasicTimer::on_button_quit()
{
    hide();
}
void BasicTimer::on_button_add_timer()
{
    if(not timer_added)
    {
        sigc::slot<bool()> my_slot = sigc::bind(sigc::mem_fun(*this,
                                                &BasicTimer::on_timeout));

        auto conn = Glib::signal_timeout().connect(my_slot,timeout_value);

        timer_added = true;

        std::cout << "Timer added" << std::endl;
    }
    else
    {
        std::cout << "The timer already exists : nothing more is created"
                  << std::endl;
    }
}
void BasicTimer::on_button_delete_timer()
{
    if(not timer_added)
    {
        std::cout << "Sorry, there is no active timer at the moment."
                  << std::endl;
    }
    else
    {
        std::cout << "manually disconnecting the timer " << std::endl;
        disconnect  = true;
        timer_added = false;
    }
}

bool BasicTimer::on_timeout()
{
    static unsigned int val(1);

    if(disconnect)
    {
        disconnect = false; // reset for next time a Timer is created

        return false; // End of Timer
    }
    // display the simulation clock
    data_label.set_text(std::to_string(val));

    std::cout << "This is simulation update number : " << val << std::endl;
    ++val;
    return true;
}
```

La nouveauté de cet exemple provient des signal handlers des Buttons Add et Delete :

- Pour **start_timer** le signal handler **on_button_add_timer()** s'assure que le booléen est à false avant de créer le Timer. Celui-ci est associé à une fonction callback **on_timeout()** qui sera automatiquement appelée chaque fois que la période de 500ms est écoulée. Cela fait on change l'état du booléen **timer_added** à vrai pour ne pas créer d'autres Timer par erreur.

- Pour **stop_timer** le signal handler **on_button_delete_timer()** s'assure qu'il y a bien un Timer en testant **timer_added**. Une demande de désactivation est mémorisée en faisant passer le booléen **disconnect** à true. Le booléen **timer_added** doit bien sûr repasser à faux pour permettre de re-créer le Timer ultérieurement avec l'autre Button.

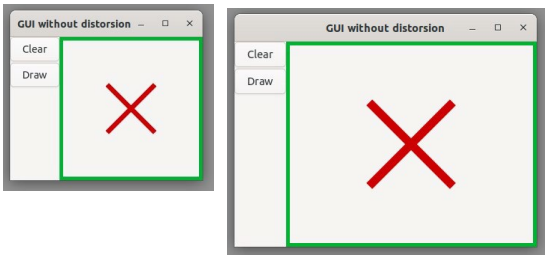
- La fonction callback **on_timeout** gère le Timer en fonction des demandes exprimées par les Buttons.

    - Si une demande de fin d'existence du Timer a été enregistrée en faisant passer le booléen **disconnect** à **true**, alors la fonction renvoie **false** ce qui a pour effet de supprimer le Timer. Auparavant il faut seulement remettre le booléen **disconnect** à **false** au cas où un Timer est créée par une demande du Button addTimer.

    - Si par contre il n'y a pas de demande de fin du Timer, on affiche avec le **Gtk ::Label data_label** la valeur d'une variable **static** locale initialisée à 1 et qui est incrémentées à chaque appel. La valeur du compteur étant un entier elle est convertie en chaîne de caractère avec la méthode **std ::to_string().** Remarquer que la fonction **on_timeout()** renvoie **true**, ce qui exprime qu'on veut que le Timer continue son activité.

Activité : changer la valeur de la période et constater la qualité de la synchronisation avec le temps du monde réel (wall-clock time). *On se satisfera de cette qualité pour le projet*. Par exemple, commencez en indiquant une période de 1000ms puis regardez votre montre au moment où vous appuyez sur Start et lorsque le programme affiche 10.

Exercice 3.(niveau 0) **: Correction de distorsion bis** Ici on dispose de la petite interface introduite la semaine précédente ; Vous trouverez les détails des explications sur la gestion d'événements provenant des Buttons dans la série5 niveau0 ; le module main.cc étant le même on ne le détaille pas ici. Pour l'absence de préservation des angles droits de la croix distorsion du dessin dans l'espace du Modèle, voir le corrigé de l'exercice 3b de la semaine dernière.



Cet exercice ajoute seulement une fonction static **draw_frame** qui dessine le cadre vert de l'espace de la fenêtre de dessin de taille width x height.

Ce dessin étant dans l'espace de la fenêtre il est effectué avant de mettre en place la transformation de coordonnées de l'espace du Modèle vers l'espace de la fenêtre avec la fonction **orthographic_projection**.

```
static void draw_frame(const Cairo::RefPtr<Cairo::Context>& cr, Frame frame)
{
      //display a rectangular frame around the drawing area
      cr->set_line_width(10.0);     // draw greenish lines
      cr->set_source_rgb(0., 0.7, 0.2);
      cr->rectangle(0,0, frame.width, frame.height);
      cr->stroke();
}
```

Exercice 4.(niveau 0) : **choisir un nom de fichier (pour la lecture / l'écriture)**

Cet exercice simplifie l'exemple FileChooserDialog fourni par le manuel GTKmm 4. C'est ce qu'il faut ré-utiliser pour ouvrir/sauvegarder un fichier.

Le programme principal définit un widget ExampleWindow dérivé de la classe Window

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

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

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

L'interface du module **exampleWindow** définit 2 Button avec leur signal handler :

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H
#include <gtkmm.h>

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

protected:
  //Signal handlers:
  void on_button_open_clicked();
  void on_button_save_clicked();
  void on_file_dialog_response(int response_id,
                               Gtk::FileChooserDialog* dialog);

  //Child widgets:
  Gtk::Box m_Box;
  Gtk::Button m_Button_Open;
  Gtk::Button m_Button_Save;
};
#endif //GTKMM_EXAMPLEWINDOW_H
```

La première partie de l'implémentation contient le constructeur. On dispose les Button et le Label verticalement dans un Gtk::Box. On connecte chaque Button à son signal handler.

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


ExampleWindow::ExampleWindow():
    m_Box(Gtk::Orientation::VERTICAL),
      m_Button_Open("Open File"),
      m_Button_Save("Save File")
{
    set_title("Gtk::FileSelection example");

    set_child(m_Box);

    m_Box.append(m_Button_Open);
    m_Button_Open.set_expand(true);
    m_Box.append(m_Button_Save);
    m_Button_Save.set_expand(true);

    m_Button_Open.signal_clicked().connect(sigc::mem_fun(*this,
                  &ExampleWindow::on_button_open_clicked) );

    m_Button_Save.signal_clicked().connect(sigc::mem_fun(*this,
                  &ExampleWindow::on_button_save_clicked) );}

ExampleWindow::~ExampleWindow()
{
}
```

L'aspect intéressant est le signal handler des Button. On présente seulement celui du button Open ; l'autre est visible dans le code source.

Cette méthode commence par déclarer une instance **dialog** de **Gtk ::FileChooserDialog** qui servira de support pour gérer le choix d'un nom de fichier dans une fenêtre temporaire (noter **Action ::OPEN**).

Cet objet de dialogue dispose d'une méthode **run()** qui va capter l'activité de l'interface, c'est-à-dire qu'on ne pourra rien faire d'autre tant que ce dialogue sera en cours.

Le caractère « temporaire » de l'instance **dialog** est spécifié par l'appel de la méthode **set_transient_for**() qui suit. Cela garantit le placement de la fenêtre de dialogue au-dessus de la fenêtre de l'application.

On doit explicitement demander quels boutons de dialogue on veut : ici on demande un button **CANCEL** et un bouton **OPEN** qui confirme le choix.

Les constantes **CANCEL** et **OK**, fournies en second paramètres, sont testées dans la méthode **on_file_dialog_response()** ; cette même méthode sert aussi pour gérer la réponse du dialogue de l'autre Button.

```
void ExampleWindow::on_button_open_clicked()
{
    auto dialog = new Gtk::FileChooserDialog("Please choose a file",
                                    Gtk::FileChooser::Action::OPEN);
    dialog->set_transient_for(*this);
    dialog->set_modal(true);
    dialog->signal_response().connect(sigc::bind(
      sigc::mem_fun(*this, &ExampleWindow::on_file_dialog_response),
      dialog));

    //Add response buttons to the dialog:
    dialog->add_button("_Cancel", Gtk::ResponseType::CANCEL);
    dialog->add_button("_Open", Gtk::ResponseType::OK);

    //Add filters, so that only certain file types can be selected:

    auto filter_text = Gtk::FileFilter::create();
    filter_text->set_name("Text files");
    filter_text->add_mime_type("text/plain");
    dialog->add_filter(filter_text);

    auto filter_cpp = Gtk::FileFilter::create();
    filter_cpp->set_name("C/C++ files");
    filter_cpp->add_mime_type("text/x-c");
    filter_cpp->add_mime_type("text/x-c++");
    filter_cpp->add_mime_type("text/x-c-header");
    dialog->add_filter(filter_cpp);

    auto filter_any = Gtk::FileFilter::create();
    filter_any->set_name("Any files");
    filter_any->add_pattern("*");
    dialog->add_filter(filter_any);

    //Show the dialog and wait for a user response:
    dialog->show();
}
```

La méthode `on_file_dialog_response` gère la réponse choisie pendant l'interaction avec ce widget.

La constante obtenue quand on clicque sur le bouton OPEN est `OK`.

Attention il n'y a pas eu d'ouverture de fichier à ce stade !

Cette réponse veut simplement dire qu'on peut *récupérer le nom de fichier choisi* comme dans cet exemple avec la variable **filename** qui contient le chemin absolu depuis la racine de l'arborescence des fichiers ; on peut le constater quand on l'affiche dans le terminal.

Activité : utiliser la variable filename pour appeler une fonction ou méthode qui ouvre ce fichier pour, par exemple, l'afficher dans le terminal.

```
void ExampleWindow::on_file_dialog_response(int response_id,
                                            Gtk::FileChooserDialog* dialog)
{
    //Handle the response:
    switch (response_id)
    {
        case Gtk::ResponseType::OK:
        {
            std::cout << "Open or Save clicked." << std::endl;

            //Notice that this is a std::string, not a Glib::ustring.
            auto filename = dialog->get_file()->get_path();
            std::cout << "File selected: " <<  filename << std::endl;
            break;
        }
        case Gtk::ResponseType::CANCEL:
        {
            std::cout << "Cancel clicked." << std::endl;
            break;
        }
        default:
        {
            std::cout << "Unexpected button clicked." << std::endl;
            break;
        }
    }
    delete dialog;
}
```

Complément : Comment passer de **OPEN** à **SAVE** ?

Le même objet **Gtk ::FileChooserDialog** sert pour les deux opérations ; il faut seulement le configurer avec un symbole différent selon l'utilisation.

Pour OUVRIR : on indique le symbole **`Gtk::FileChooser::Action::OPEN`** dans la déclaration du **Gtk ::FileChooserDialog** :

```
Gtk::FileChooserDialog dialog("Please choose a file",
                      Gtk:: FileChooser::Action::OPEN);
```

Et on ajoute le Button de dialogue qui signale l'action **_Open** dans la fenêtre de dialogue :

```
dialog->add_button("_Open", Gtk::ResponseType::OK);
```

Pour SAUVEGARDER : on indique le symbole **`Gtk::FileChooser::Action::SAVE`** dans la déclaration du **Gtk ::FileChooserDialog** :

```
Gtk::FileChooserDialog dialog("Please choose a file",
                      Gtk::FileChooser::Action::SAVE);
```

Et on ajoute le Button de dialogue qui signale l'action **_Save** (au lieu de **_Open**) dans la fenêtre de dialogue :

```
dialog->add_button("_Save", Gtk::ResponseType::OK);
```