PoP Série 5 niveau 0

Usage de GTKmm pour l'interface graphique utilisateur

Code : [Layout, gestion d'evenement, timer et sélection de fichier]

Exercice 1.1(niveau 0) : **layout et variable d'état**

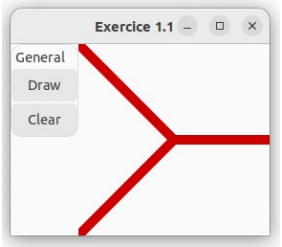
Dans cet exemple nous allons avoir une interaction entre les boutons et l'affichage.

1.1.1) Le programme principal définit un widget de la classe MyEvent. Il suit le même modèle que les exemples de la semaine dernière. Voici la fenêtre créée à l'exécution :

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

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

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



1.1.2) La classe **MyEvent** contient les éléments pour gérer des boutons et du dessin :

La structure du GUI est définie avec des « boîtes » Gtk ::Box et un cadre de type Gtk ::Frame.

On reconnait deux boutons et un outil de dessin =>

La variable d'état « draw » sert à communiquer les intentions exprimées avec les boutons à la méthode on_draw()

```
10  class MyEvent : public Gtk::Window
11  {
12  public:
13      MyEvent();
14
15  private:
16      // GUI layout
17      Gtk::Box m_Main_Box;
18      Gtk::Box m_Panel_Box;
19      Gtk::Box m_Buttons_Box;
20      Gtk::Frame m_Panel_Frame;
21      Gtk::Button m_Button_Draw;
22      Gtk::Button m_Button_Clear;
23      Gtk::DrawingArea m_Area;
24
25      bool draw ; // current drawing state
26
27      //Button Signal handlers:
28      void on_button_clicked_clear();
29      void on_button_clicked_draw();
30
31      // DrawingArea signal handler:
32      void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
33                   int width, int height);
34  };
```

L'implémentation myevent.cc définit d'abord une constante puis le constructeur. Celui-ci définit tout d'abord des options d'organisation HORIZONTAL ou VERTICAL des attributs **Box** puis les noms des attributs **Frame** et **Button** dans la liste d'initialisation. En particulier, l'attribut **m_Main_Box** est initialisé avec l'option HORIZONTAL, ce qui veut dire que les éléments qu'il contiendra seront alignés sur une même ligne horizontale. Par contre l'attribut **m_Buttons_Box** est de type VERTICAL, c'est pourquoi les boutons qu'il contiendra sont alignés dans une colonne.

```
 8  MyEvent::MyEvent():
 9      m_Main_Box(Gtk::Orientation::HORIZONTAL, 0),
10      m_Panel_Box(Gtk::Orientation::VERTICAL, 2),
11      m_Buttons_Box(Gtk::Orientation::VERTICAL, 2),
12      m_Panel_Frame("General"),
13      m_Button_Draw("Draw"),
14      m_Button_Clear("Clear"),
15      draw(true)
16  {
17      // init layout
18      set_title("Exercice 1.1");
19      set_child(m_Main_Box);
20
21      m_Main_Box.append(m_Panel_Box);
22      m_Main_Box.append(m_Area);
23
24      m_Panel_Box.append(m_Panel_Frame);
25      m_Panel_Frame.set_child(m_Buttons_Box);
26      m_Buttons_Box.append(m_Button_Draw);
27      m_Buttons_Box.append(m_Button_Clear);
28
29      // init buttons signal handlers
30      m_Button_Clear.signal_clicked().connect(
31          sigc::mem_fun(*this, &MyEvent::on_button_clicked_clear));
32
33      m_Button_Draw.signal_clicked().connect(
34          sigc::mem_fun(*this, &MyEvent::on_button_clicked_draw));
35
36      // init drawing sub-window parameters
37      m_Area.set_content_width(area_side);
38      m_Area.set_content_height(area_side);
39      m_Area.set_expand();
40      m_Area.set_draw_func(sigc::mem_fun(*this, &MyEvent::on_draw));
41
42  }
```

Ensuite, dans le corps du constructeur, on définit le titre de la fenêtre et on indique que la boîte **m_Main_Box** est le conteneur à la racine de l'interface graphique**.** Ensuite, lignes 21-22, on y ajoute avec la méthode **append** le widget de la boîte des boutons **m_Panel_Box** suivi *horizontalement* par le widget de dessin **m_Area**

On passe ensuite au remplissage du conteneur **m_Panel_Box** avec la méthode **append**. On y ajoute un élément **Gtk ::Frame** qui sert de cadre avec un texte de titre (« General »).Ligne 24, le **Gtk ::Frame** sert de racine pour la boîte **m_Buttons_Box** contenant les 2 boutons qui vont apparaitre verticalement conformément à son initialisation.

Ensuite, lignes 30-34, on initialise les signal handlers des 2 boutons ; chaque **Button** fournit l'adresse d'une méthode de la classe **MyEvent**. Leur action est discutée plus loin.

La fin du constructeur, lignes 37-40, initialise des paramètres de la zone de dessin : sa taille, lignes 37-38, l'autorisation de changer de taille (ligne 39), et, ligne 40, la connexion de son signal handler **on_draw** qui redessine son contenu dès qu'un événement de rafraichissement du dessin est détecté (ex : déplacer la fenêtre ou changer sa taille).

Le point important à observer concernant **on_draw** est que nous *n'avons pas le droit de changer le prototype de cette méthode.* Or, on aurait bien aimé pouvoir la paramétrer en fonction du but du programme et/ou de l'action des boutons. Comme cela n'est pas possible on utilise la notion de variable d'état, ici avec l'attribut **draw**, qui mémorise la demande exprimée par l'action des boutons. La valeur courante de cet attribut **draw** est ensuite utilisée dans **on_draw** pour déterminer ce qui est dessiné : cela est possible sans passage de paramètre car l'attribut **draw** et la méthode **on_draw** appartiennent à la même portée de classe **MyEvent**. Voici d'abord le code des signal handlers des boutons :

```
44  void MyEvent::on_button_clicked_clear()
45  {
46      std::cout << "Drawing cancelled" << std::endl;
47      draw = false;
48      m_Area.queue_draw();
49  }
50
51  void MyEvent::on_button_clicked_draw()
52  {
53      std::cout << "Drawing activated" << std::endl;
54      draw = true;
55      m_Area.queue_draw();
56  }
```

Les deux points importants à observer sont 1) les changements effectués sur la variable d'état **draw** (lignes 47 et 54) et 2) l'appel de la méthode **queue_draw()** sur l'attribut de dessin **m_Area** (lignes 48 et 55). En effet, avec GTKmm, comme on ne peut pas directement appeler **on_draw()** sur l'attribut **m_Area**, à la place on produit un événement de demande de mise à jour du dessin qui sera traité automatiquement par GTKmm par un appel de **on_draw()**. On dispose donc d'un moyen *indirect* d'appel de **on_draw().** Sachant que l'événement est dans la file d'attente, GTKmm va d'abord traiter d'éventuels autres événements qui seraient plus anciens que vos demandes des lignes 48 et 55. Voici le code de **on_draw**() :

```
58  void MyEvent::on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
59                        int width, int height)
60  {
61      if(draw)
62      {
63          // coordinates for the center of the window
64          int xc, yc;
65          xc = width / 2;
66          yc = height / 2;
67
68          cr->set_line_width(10.0);
69
70          // draw red lines out from the center of the window
71          cr->set_source_rgb(0.8, 0.0, 0.0);
72          cr->move_to(0, 0);
73          cr->line_to(xc, yc);
74          cr->line_to(0, height);
75          cr->move_to(xc, yc);
76          cr->line_to(width, yc);
77          cr->stroke();
78      }
79      else
80      {
81          std::cout << "Nothing to draw !" << std::endl;
82      }
83  }
```

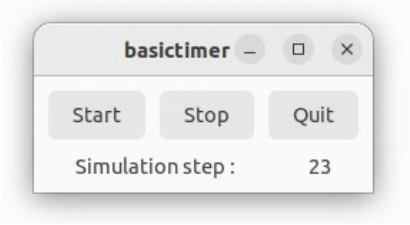
Activité :

1) inversez les options HORIZONTAL et VERTICAL pour les Box et constatez le résultat
2) En plus, changez aussi l'ordre des éléments dans m_Main_box, c'est-à-dire ajoutez d'abord m_Panel_Box avant m_Area

Exercice 1.2 (niveau 0) : **mise en œuvre d'un timer**

Dans cet exemple nous allons activer/désactiver un timer et décider de l'action à réaliser à chaque événement créé par ce timer. C'est l'outil idéal pour faire évoluer le jeu en continu.

1.2.1 L'interface de la classe **BasicTimer** contient les éléments pour gérer les boutons, faire afficher du texte (Label), et bien créer/détruire un timer :




```
 8  class BasicTimer : public Gtk::Window
 9  {
10  public:
11      BasicTimer();
12
13  private:
14      Gtk::Box m_box, top_box,   bottom_box;
15      Gtk::Button     start_timer, stop_timer, quit_button;
16      Gtk::Label      text_label, data_label;
17
18      // state variable to handle a single timer
19      bool timer_added;
20      // state variable for the timer disconnect request
21      bool disconnect;
22      // duration initialized in constructor
23      const int timeout_value;
24
25      // button signal handlers
26      void on_button_add_timer();
27      void on_button_delete_timer();
28      void on_button_quit();
29
30      // Timer signal handler
31      bool on_timeout();
32  };
```

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.

On utilise des Labels dans la seconde ligne de la fenêtre pour documenter l'exécution avec **text_label** avec du texte constant (Simulation step :) tandis que **data_label** va montrer la valeur courante d'un compteur.

La Box **m_box** sert de racine pour organiser l'interface ; les boutons sont disposés horizontalement dans la Box **top_box**, tandis que les labels sont disposés aussi horizontalement mais dans la Box **bottom_box** placée verticalement sous la **top_box**.

Les deux variables d'état mises à jour par les boutons sont :

- Le booléen **timer_added** mémorise 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 est initialisée dans le constructeur (à 500 ms), visible page suivante.

1.2.1 L'implémentation de la classe **BasicTimer** commence avec son constructeur qui définit un contexte *sans* timer (le booléen **timer_added** est false). Le constructeur se contente d'organiser les boutons et les labels et de connecter les boutons à leur signal handler.

```
4   BasicTimer::BasicTimer() :
5       m_box(Gtk::Orientation::VERTICAL, 10),
6       top_box(Gtk::Orientation::HORIZONTAL, 10),
7       bottom_box(Gtk::Orientation::HORIZONTAL, 10),
8       start_timer("_Start", true),
9       stop_timer("_Stop", true),
10      quit_button("_Quit", true),
11      text_label("Simulation step : "),
12      data_label("0"),
13      timer_added(false),// to handle a single timer
14      disconnect(false), // to handle a single timer
15      timeout_value(500) // 500 ms = 0.5 seconds
16  {
17      m_box.set_margin(10);
18      set_child(m_box);
19
20      m_box.append(top_box);
21      m_box.append(bottom_box);
22
23      top_box.append(start_timer);
24      top_box.append(stop_timer);
25      top_box.append(quit_button);
26
27      start_timer.set_expand();
28      stop_timer.set_expand();
29      quit_button.set_expand();
30
31      bottom_box.append(text_label);
32      bottom_box.append(data_label);
33
34      text_label.set_expand();
35      data_label.set_expand();
36
37      // Connect the three buttons:
38      quit_button.signal_clicked().connect(sigc::mem_fun(*this,
39                &BasicTimer::on_button_quit));
40      start_timer.signal_clicked().connect(sigc::mem_fun(*this,
41                &BasicTimer::on_button_add_timer));
42      stop_timer.signal_clicked().connect(sigc::mem_fun(*this,
43                &BasicTimer::on_button_delete_timer));
44  }
```

Le signal handler du timer **on_timeout()** n'est pas utilisé dans le constructeur. Cela est fait lignes 58-59 dans le signal handler du bouton **start_timer** quand on clique dessus ; on indique aussi la durée ligne 62 avec timeout_value. La variable d'état **timer_added** est testée et mise à jour pour réagir de manière cohérente :

```
51  void BasicTimer::on_button_add_timer()
52  {
53      if(not timer_added)
54      {
55          // Creation of a new object prevents long lines and shows us a little
56          // how slots work.  We have 0 parameters and bool as a return value
57          // after calling sigc::bind.
58          sigc::slot<bool()> my_slot = sigc::bind(sigc::mem_fun(*this,
59                                                  &BasicTimer::on_timeout));
60
61          // This is where we connect the slot to the Glib::signal_timeout()
62          auto conn = Glib::signal_timeout().connect(my_slot,timeout_value);
63
64          timer_added = true;
65
66          std::cout << "Timer added" << std::endl;
67      }
68      else
69      {
70          std::cout << "The timer already exists : nothing more is created"
71                    << std::endl;
72      }
73  }
```

Le signal handler du bouton **stop_timer** travaille avec les 2 variables d'état : **timer_added** lui permet de vérifier s'il y a effectivement un timer à supprimer, et si c'est le cas cette méthode se contente de mémoriser la demande avec l'autre variable d'état **disconnect** qu'on fait passer à **true**. En effet, seule le signal handler du timer lui-même, c'est-à-dire **on_timeout**(), peut le désactiver ; ici on se contente de mémoriser un changement d'état dans l'attribut **disconnect** car celui-ci sera visible par l'autre signal handler quand il sera appelé.

```
75  void BasicTimer::on_button_delete_timer()
76  {
77      if(not timer_added)
78      {
79          std::cout << "Sorry, there is no active timer at the moment." << std::endl;
80      }
81      else
82      {
83          std::cout << "manually disconnecting the timer " << std::endl;
84          disconnect  = true;
85          timer_added = false;
86      }
87  }
```

Voici le signal handler du timer ; il a 2 tâches à effectuer. La première est de décider s'il doit continuer à exister ou pas et il réagit à la valeur de la variable d'état **disconnect**. Si elle est vraie alors le timer s'arrête ; il l'indique à GTKmm en renvoyant **false**. Sinon, on effectue la tâche prévue : ici on met à jour l'affichage du Label **data_label** avec une conversion du compteur **val** vers string et on incrémente ce compteur. Il est important de renvoyer **true** qui confirme que le timer doit produire au moins un autre événement.

```
89  bool BasicTimer::on_timeout()
90  {
91      static unsigned int val(1);
92
93      if(disconnect)
94      {
95          disconnect = false; // reset for next time a Timer is created
96
97          return false; // End of Timer
98      }
99
100     data_label.set_text(std::to_string(val));  // display the clock
101
102     std::cout << "This is simulation update number : " << val << std::endl;
103
104     ++val;
105     return true;
106 }
```

Cet exemple d'exécution montre l'effet de l'action sur les boutons et les vérifications qui sont faites. Lorsque le timer est actif le signal handler **on_timeout**() est appelé tous les 500 ms et produit l'affichage de la valeur courante de **val** dans le terminal (et aussi dans la fenetre GTK grâce au Label).

Activité : essayer d'autres valeurs de **timeout_value** (ligne 15). C'est en ms.

Pour le projet on recommande d'indiquer 25 ms comme intervalle de temps entre deux événements du timer pour une bonne réactivité des entités du jeu tout en ayant suffisamment de temps pour les calculs d'une mise à jour du jeu.

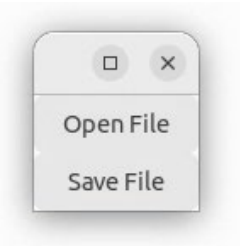
```
KBasicTimer$ ./basictimer
libEGL warning: DRI2: failed to authentica
te
Timer added
This is simulation update number : 1
This is simulation update number : 2
This is simulation update number : 3
This is simulation update number : 4
This is simulation update number : 5
This is simulation update number : 6
This is simulation update number : 7
This is simulation update number : 8
This is simulation update number : 9
manually disconnecting the timer
Timer added
This is simulation update number : 10
This is simulation update number : 11
The timer already exists : nothing more is
 created
This is simulation update number : 12
This is simulation update number : 13
This is simulation update number : 14
manually disconnecting the timer
```

Exercice 1.3 (niveau 0) : **sélection de fichiers**

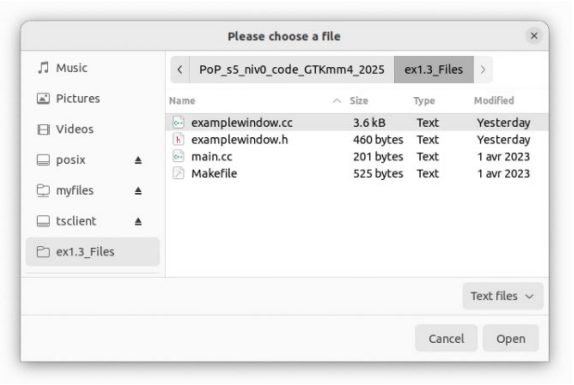
Cet exemple montre comment indiquer la chaine de caractères d'un nom de fichier existant (pour la lecture) ou à créer (pour la sauvegarde).

1.2.1 L'interface de la classe **ExampleWindow** contient les éléments pour gérer les 2 boutons (lignes 14-15) avec leur signal handlers (lignes 21-22), ainsi que le signal handler **on_file_dialog_response** (ligne 23) d'une fenêtre de dialogue automatiquement ouverte quand on utilise les boutons.

```
9   public:
10    ExampleWindow();
11
12  private:
13    //Child widgets:
14    Gtk::Box m_Box;
15    Gtk::Button m_Button_Open;
16    Gtk::Button m_Button_Save;
17
18    std::string filename;
19
20    //Signal handlers:
21    void on_button_open_clicked();
22    void on_button_save_clicked();
23    void on_file_dialog_response(int response_id, Gtk::FileChooserDialog* dialog);
24  };
```



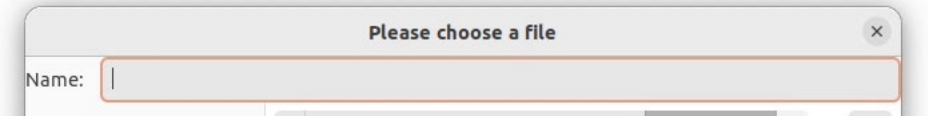
Les signal handlers des boutons contiennent beaucoup d'instructions pour configurer la fenêtre de dialogue (ex : lignes 39-40) ainsi que des filtres d'affichage. L'important est de remarquer qu'il lance une fenêtre de dialogue et surtout qu'il lui associe un autre signal handler (ex : ligne 36) qui traite la réponse de la personne qui utilise cette fenêtre de dialogue.




```
29  void ExampleWindow::on_button_open_clicked()
30  {
31      auto dialog = new Gtk::FileChooserDialog("Please choose a file",
32            Gtk::FileChooser::Action::OPEN);
33      dialog->set_transient_for(*this);
34      dialog->set_modal(true);
35      dialog->signal_response().connect(sigc::bind(
36      sigc::mem_fun(*this, &ExampleWindow::on_file_dialog_response), dialog));
37
38      //Add response buttons to the dialog:
39      dialog->add_button("_Cancel", Gtk::ResponseType::CANCEL);
40      dialog->add_button("_Open", Gtk::ResponseType::OK);
```

…

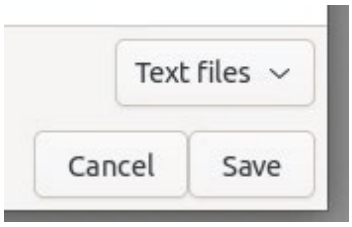
Le signal handler du bouton Save est presque le même que celui du bouton Open. On change seulement « open » par « save » pour les lignes 65, 68 et 76. La différence principale de sa fenêtre de dialogue est la zone encadrée en orange ci-dessous dans laquelle on doit écrire soi-même le nom désiré de fichier pour qu'il soit ensuite créé avec une opération de sauvegarde. Le reste de la fenêtre sert à choisir le répertoire de destination.

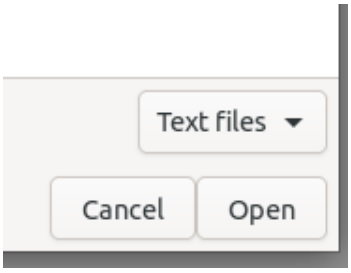


C'est le signal handler de la fenêtre de dialogue **on_file_dialog_response** qui traite la réponse du dialogue => **response_id** obtenue en paramètre.

Les deux cas avec succès (clic sur le bouton Open ou sur Save) donnent la valeur OK (ligne 108) qui permet d'affecter à l'attribut **filename** (ligne 113) le chemin complet indiquant le nom de fichier, c'est-à-dire avec la chaîne des répertoires parents.

Si par contre on a cliqué sur le bouton Cancel on obtient la valeur CANCEL (ligne 117) et l'attribut **filename** n'est pas mis à jour.






```
102 void ExampleWindow::on_file_dialog_response(int response_id,
103                                             Gtk::FileChooserDialog* dialog)
104 {
105     //Handle the response:
106     switch (response_id)
107     {
108         case Gtk::ResponseType::OK:
109         {
110             std::cout << "Open or Save clicked." << std::endl;
111
112             //Notice that this is a std::string, not a Glib::ustring.
113             filename = dialog->get_file()->get_path();
114             std::cout << "File selected: " <<  filename << std::endl;
115             break;
116         }
117         case Gtk::ResponseType::CANCEL:
118         {
119             std::cout << "Cancel clicked." << std::endl;
120             break;
121         }
122         default:
123         {
124             std::cout << "Unexpected button clicked." << std::endl;
125             break;
126         }
127     }
128     delete dialog;
129 }
```

Activité : se familiariser avec le fonctionnement de la fenêtre de dialogue.

Pour le projet, l'attribut du nom de fichier doit ensuite être passé à la méthode du Modèle qui va lire ce fichier.