Architecture d'un programme interactif graphique

Partie 2: Programmation par événement

Objectifs:

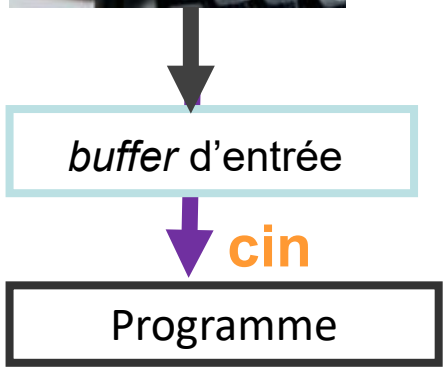
- maîtriser le concept de programmation par événement

Plan:

- Principe de la lecture non-bloquante
- Programmation par événements
- Exemple détaillé myEvent (complété dans la série5 niv0)

Principe de la lecture non-bloquante => programmation par événements

Entrée conversationnelle
= lecture bloquante



Tant qu'on n'a pas validé ce qui est tapé au clavier avec **Enter**, le *buffer* d'entrée est vide :

→ il n'y a rien à «lire» pour le programme

→ Le programme attend…

Programmation par événements
= lecture non-bloquante

L'essentiel du temps d'exécution est passé pendant l'exécution de la méthode **run** sur l'Application **app** de GTKmm.

```
…
int main(…)
{
  auto app = Gtk::Application …
  …
  return app->... ;
}
```

La gestion de l'interaction est non-bloquante car la méthode **run** gère une boucle infinie de traitement des ***événements***.

L'***événement*** est l'atome de l'interaction.

C'est un **changement d'état** d'un élément de l'interface GTKmm, y compris de la fenêtre graphique, du clavier et des boutons de la souris,

Dans la terminologie de **GTKmm** à chaque ***événement*** correspond la production d'un ***signal*** spécifique

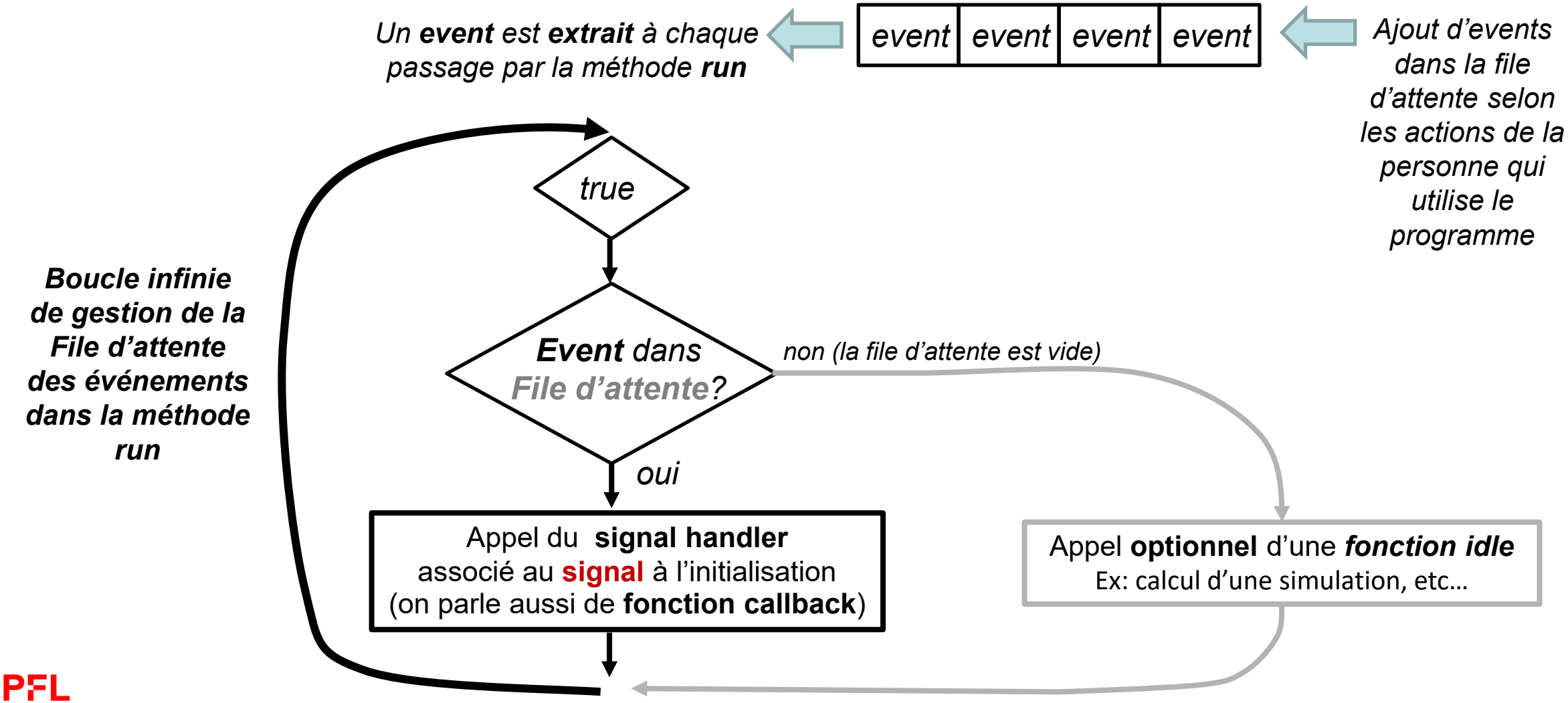
Ex: appuyer sur un `Button` produit `signal_clicked`

Si la classe dérivée de `Window` a initialisé **un signal handler**, celui-ci est appelé automatiquement

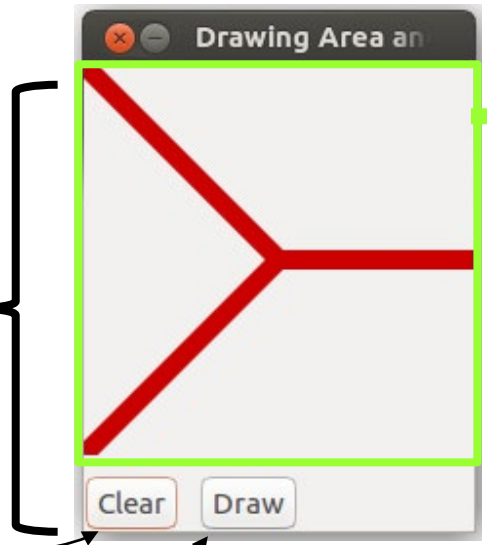
Principe de la lecture non-bloquante => programmation par événements (2)

Pseudocode de la boucle infinie de gestion de la file d'attente des événements dans la méthode run(…)

Chaque ***événement*** **=** ***event*** **=** ***signal*** est mémorisé dans une **File d'attente d'événements** selon son instant de création



Répartition des rôles dans l'exemple myEvent.cc



1

L'interface **myEvent** est dérivée de **Window**

Elle contient en particulier :

- Les attributs
  - **m_Area** pour le dessin
  - **m_Button_clear** pour effacer **m_Area**
  - **m_Button_draw** pour redessiner **m_Area**
- Un **signal handler** pour chaque m_Button
  - **on_button_clicked_clear()**
  - **on_button_clicked_draw()**

2

myArea est dérivée de **DrawingArea**

En plus de la redéfinition de **on_draw()**, on trouve:

- Un attribut booléen **empty**
- Des méthodes:
  - **clear()**
  - **draw()**

3 Comment ça marche ?

L'attribut booléen **empty** joue le rôle d'une ***boite à lettre***

- ***false*** => **on_draw()** dessine le motif rouge
- ***true*** => **on_draw()** envoie seulement un message sur **cout**

clic! Draw ⇨ Appel du signal handler : **on_button_clicked_draw()** ⇨ Appel de : **draw()** ⇨ 1) **empty** passe à ***false*** 2) **queue_draw()** ⇒

clic! Clear ⇨ Appel du signal handler : **on_button_clicked_clear()** ⇨ Appel de : **clear()** ⇨ 1) **empty** passe à ***true*** 2) **queue_draw()** ⇒

La méthode **queue_draw()** produit un **event** qui va causer l'appel de **on_draw()**

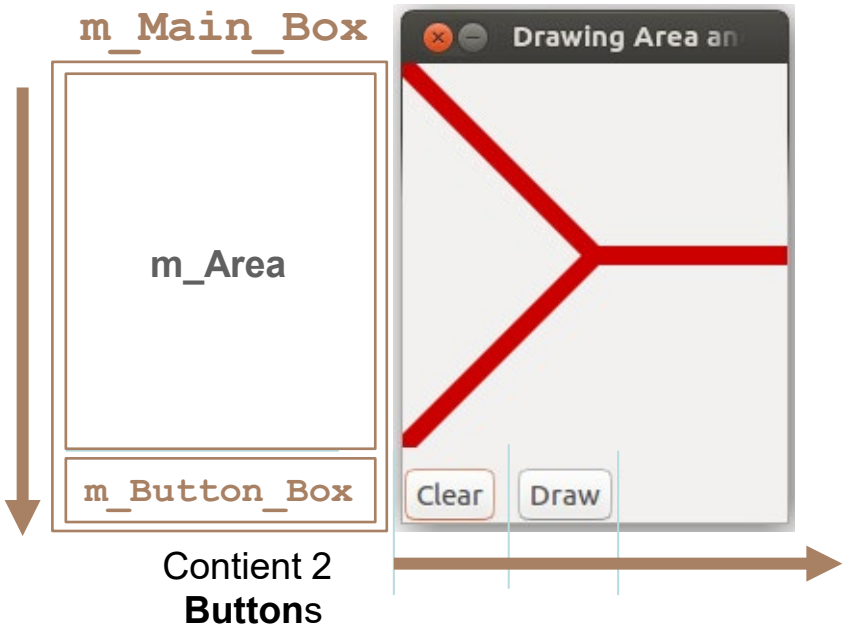
Construction de l'interface graphique à l'aide de conteneurs **Box**

Pour définir une interface graphique: **dériver** un widget de **Window**

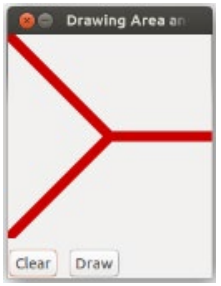
- Ce widget **possède** des attributs dont, par ex. : Buttons, **MyArea**, Box…
- Le widget **MyArea** est **dérivé** de **DrawingArea**

La mise en page est obtenue avec le conteneur **Gtk::Box**

- Un conteneur peut contenir d'autres conteneurs
- A sa création on précise si l'ajout du contenu se fait **horizontalement** ou **verticalement**



Exemple MyEvent(1)



myevent.h

```
…
class MyEvent : public Gtk::Window
{
public:
  MyEvent();

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

  Gtk::Box m_main_Box, m_button_Box;
  MyArea       m_Area;
  Gtk::Button  m_Button_Clear;
  Gtk::Button  m_Button_Draw;
};
…
```

En vert et bleu les nouveautés
par rapport à l'exemple du cours précédent

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

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

private:
  bool empty;
};
…
```

Exemple MyEvent(2)

myevent.cc

Partie layout dans le constructeur de myEvent

Mise en place des éléments du GUI à l'aide des conteneurs Box

```
  m_Main_Box(Gtk::Orientation::VERTICAL,0),
  m_Button_Box(Gtk::Orientation::HORIZONTAL,2),…   // dans la liste d'initialisation
…
  set_child(m_Main_Box);

  //Fill the main box:
  m_Main_Box.append(m_Area);
  m_Main_Box.append(m_Button_Box);

  //Fill the lower Button box
  m_Button_Box.append(m_Button_Clear);// keep fixed width by default
  m_Button_Box.append(m_Button_Draw);// keep fixed width by default
…
```

Exemple MyEvent(3)

myevent.cc / **code du widget de dessin**

```
…
void MyArea::clear()
{
  empty = true;
  queue_draw();
}

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

```
…
void MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
                     int width, int height)
{
  if(not empty)
  {
        // ici dessin comme avant
  }
  else
  {
          cout << "Empty !" << endl;
  }

}
…
```

La méthode queue_draw() produit un event = ***signal*** qui lui-même produira l'appel de **on_draw()**

Exemple MyEvent(4)

myevent.cc / **signal handlers des Buttons dans myEvent**

Partie «réactive» / mise en place des liens entre les boutons et des méthodes du widget de dessin

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

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

Résumé

- **Le code d'une application graphique interactive est structuré par une boucle infinie de traitement de la file d'attente des événements dont GTKmm a le contrôle.**
- **La conception d'une interface avec GTKmm et plus généralement avec la programmation par événements implique de réfléchir différemment à la *manière de transmettre l'information* entre fonctions ou methodes.**
- **Le passage de paramètres habituel n'est pas toujours possible car les signal handlers (fonctions callback) doivent respecter certains prototypes.**
- **Certains attributs peuvent jouer le role de relai / boite à lettre pour indiquer des changements d'état de l'application.**
- **on appelle queue_draw() pour forcer (indirectement) une demande de dessin**