

Architecture d'un programme interactif graphique

Partie 2: Programmation par événement

Objectifs:

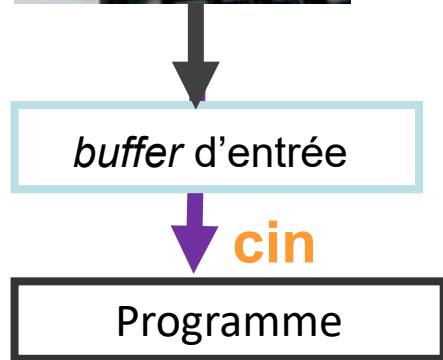
- maîtriser le concept de programmation par événement

Plan:

- Principe de la lecture non-bloquante
- Programmation par événements
- Exemple détaillé myEvent

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->run(...);  
}
```

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

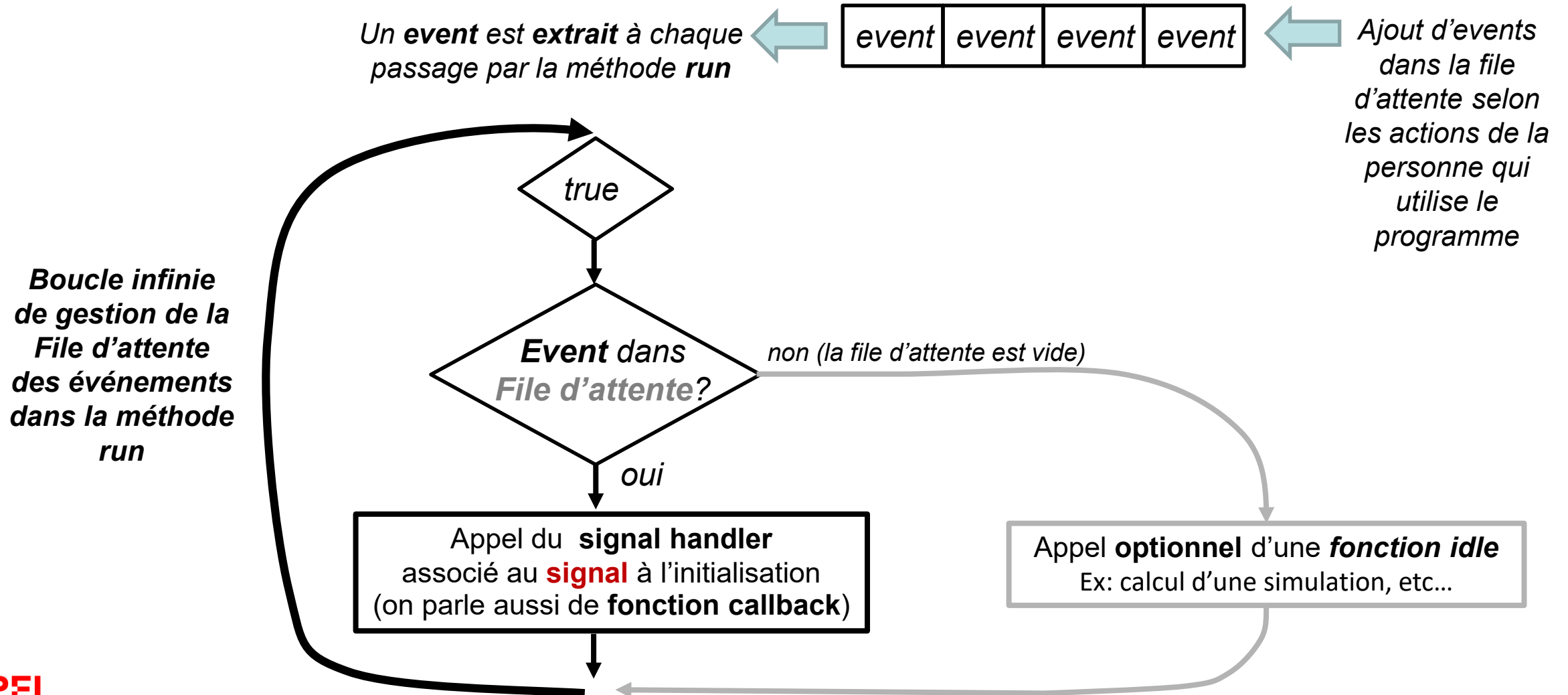
Ex: appuyer sur un **Button** produit **signal_clicked**

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



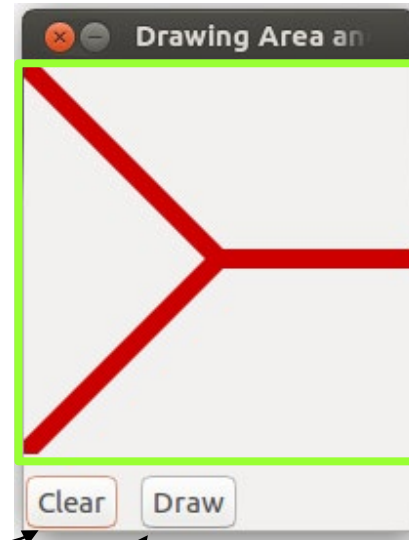
Répartition des rôles dans l'exemple myEvent.cc

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:

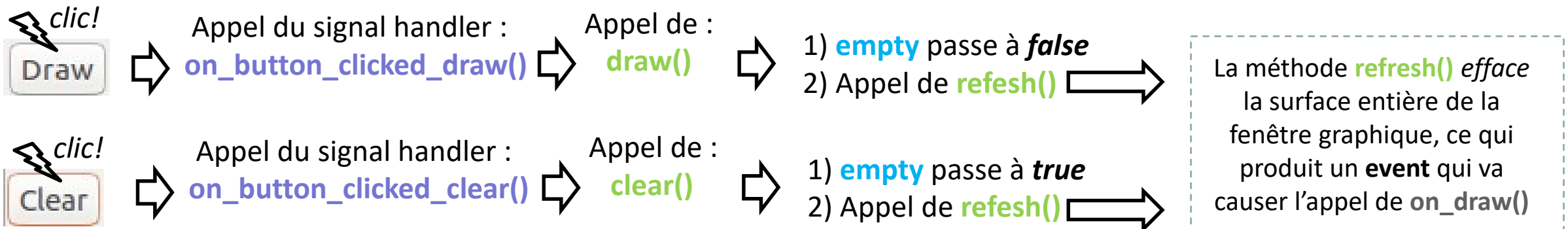
- Un attribut booléen **empty**
- Des méthodes:
 - **clear()**
 - **draw()**
 - **refresh()**

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**



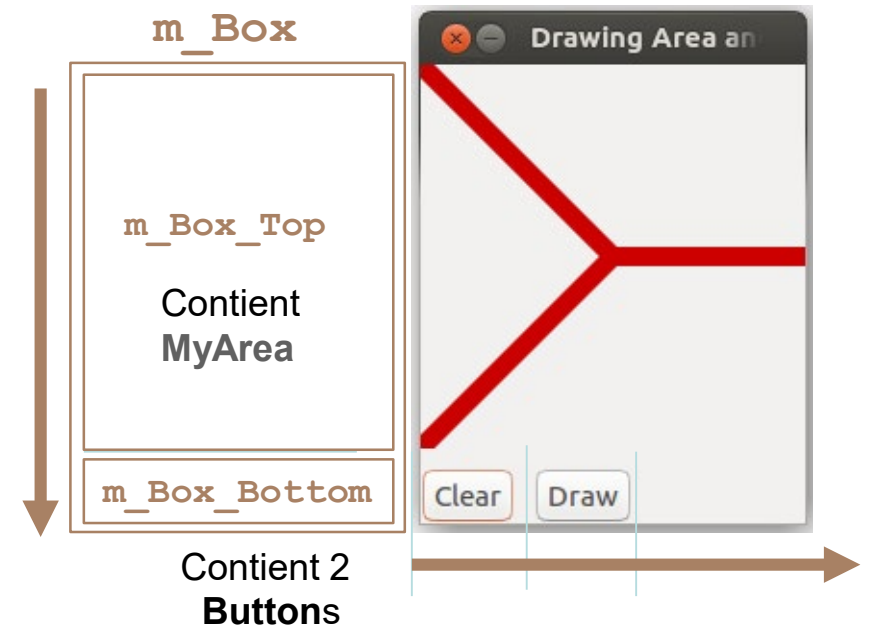
Construction de l'interface graphique à l'aide de conteneurs **Box**

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**

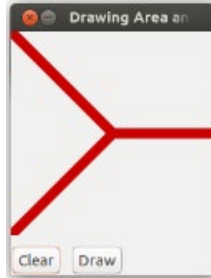
La mise en page est obtenue avec le conteneur **Gtk::Box**

- 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();
    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;
};
...
```

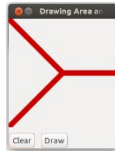
En vert et bleu les nouveautés
par rapport à l'exemple du cours précédent

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

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

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

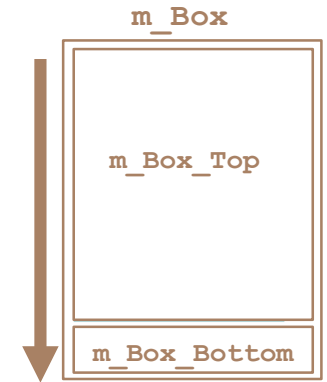
Exemple MyEvent(2)



myevent.cc

Partie layout dans le constructeur de myEvent

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



```
m_Box(Gtk::ORIENTATION_VERTICAL, 10),  
m_Box_Top(Gtk::ORIENTATION_HORIZONTAL, 10), // dans la liste d'initialisation  
m_Box_Bottom(Gtk::ORIENTATION_HORIZONTAL, 10), ...  
...  
// Add outer box to the window  
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); // permet de contrôler la taille de l'espace du dessin  
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;  
...  
...
```

Exemple MyEvent(3)

myevent.cc / code du widget de dessin



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

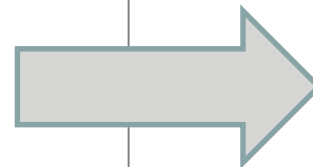
void MyArea::draw()
{
    empty = false;
    refresh();
}

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

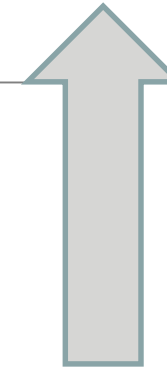
        win->invalidate_rect(r,false);
    }
}
```

```
...
bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
    if(not empty)
    {
        // ici dessin comme avant
    }
    else
    {
        cout << "Empty !" << endl;
    }

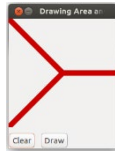
    return true;
}
...
```



Cette méthode 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 méthodes.
- 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 rôle de **relai / boîte à lettre** pour indiquer des changements d'état de l'application.
- On peut forcer indirectement un rafraichissement du dessin avec **on_draw()** en effaçant tout ou partie de l'espace de dessin avec **invalidate_rect()**.