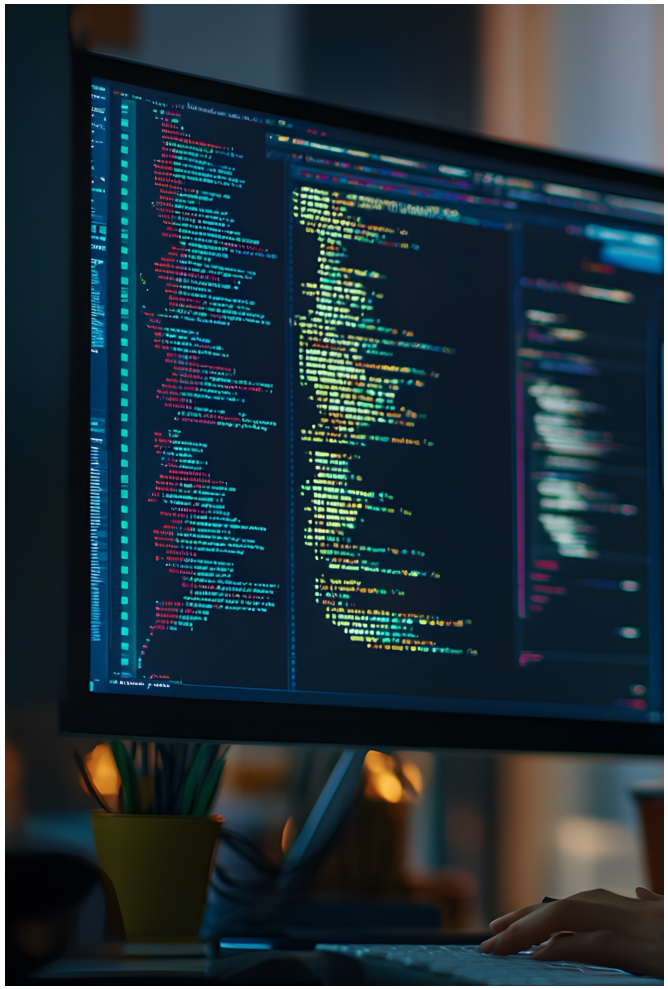
Programmation Orientée Projet
COM-112(a)

Semaine 8

Rafael Pires
rafael.pires@epfl.ch

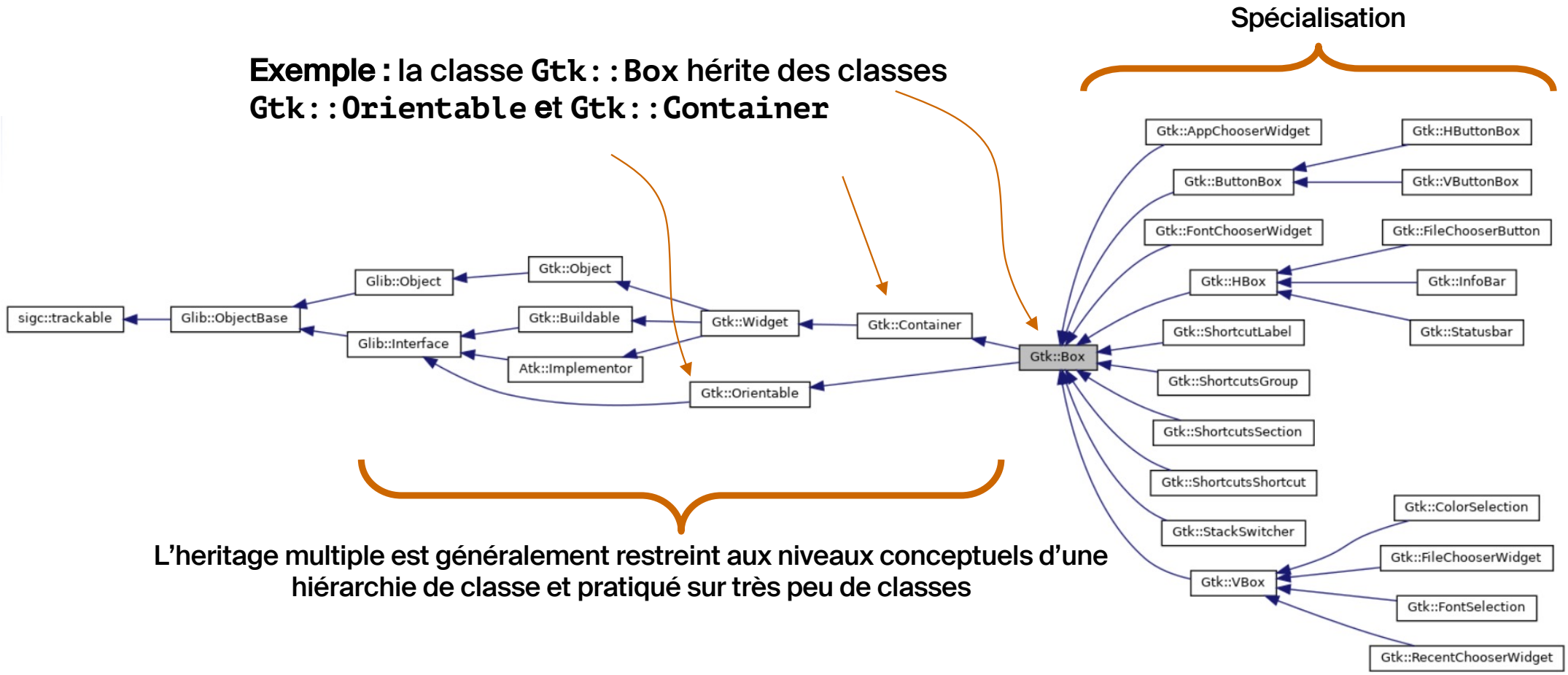
Lausanne, 17.04.2026

PoP 07



- MOOC :
  - Héritage multiple
- Projet :
  - Gtkmm4 : clavier et souris

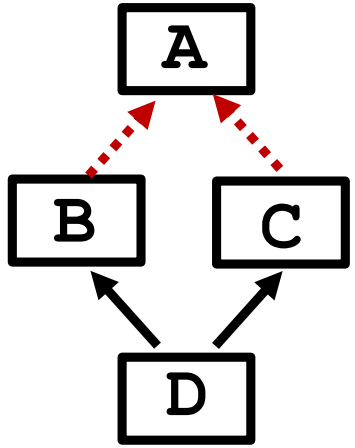
L'héritage multiple en pratique : l'exemple de GTKmm



Déclaration de l'héritage multiple

Indiquer à la déclaration la liste des classes dont on hérite :

Exemple: `class D : public B, public C { };`



L'indication d'un lien **`virtual`** est généralement nécessaire au niveau supérieur des classes parentes (ici B et C) pour éviter la duplication des attributs provenant de la classe A :

Exemple:

```
class B : public virtual A {};
class C : public virtual A {};
```

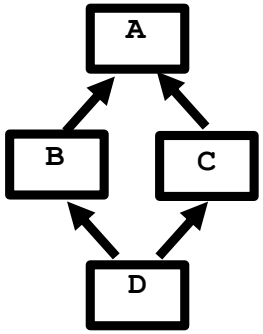
Points à surveiller : ordre des constructeurs, constructeur par défaut

Héritage multiple sans `virtual` : duplication des attributs

sans_virtual

```
class A
{
public:
  A():a(0) {}
  int obsene;
private:
  int a;
};

class B : public A
{
};

class C : public A
{
};

class D : public B, public C
{
};

int main()
{
  D d1;
  cout << d1.B::obsene << endl;
  cout << d1.C::obsene << endl;
}
```

Cet exemple syntaxiquement correct, **sans lien `virtual`**, illustre la **duplication de l'attribut** dans la classe D

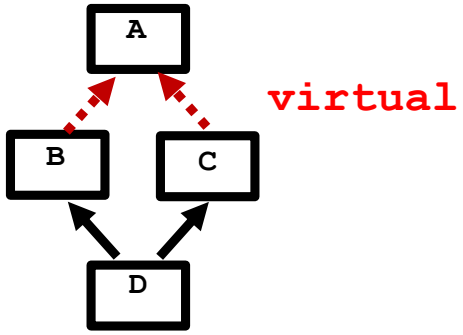


Affichage possible à l'exécution:

```
4196832
4196240
```

Héritage virtuel : c'est la classe la plus dérivée qui construit A

```
class A
{
public:
  A() : a(33) {}
  A(int x): a(x){}
  void afficher(){cout << a << endl;}
private:
  int a;
};

class B : public virtual A
{
public:
  B() : A(1) {}
};

class C : public virtual A
{
public:
  C() : A(2) {}
};

class D : public B, public C {};

int main()
{
  D d1;
  d1.afficher();
  return 0;
}
```



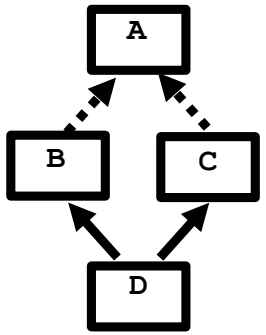
Examinons la déclaration de **d1** dans **main()**:

- Aucune valeur initiale fournie
- Constructeur par défaut par défaut de D
- Du fait des liens `virtual` entre les classes **B** et **C** et leur superclasse **A**, => **appel explicite du constructeur par défaut de A**, avant l'appel par défaut des constructeurs de B puis de C.
- Pour la même raison, **PAS d'appels aux constructeurs de A depuis les constructeurs par défaut de B et de C**

Compile et affiche 33 à l'exécution

Héritage multiple : Quiz 1

```
class A
{
public:
  A(int x) : a(x) {}
private:
  int a;
};

class B : public virtual A
{
public:
  B() : A(0) {}
};

class C : public virtual A
{
public:
  C() : A(1) {}
};

class D : public B, public C
{
};

int main()
{
  D d1;
  return 0;
}
```



Question1 : ce code…

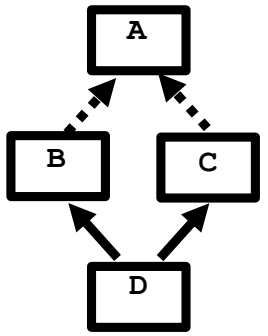
A compile et s'exécute sans problème (il ne fait rien, le programme se termine immédiatement)

B compile mais exécution avec segmentation fault

C ne compile pas

Héritage multiple : Quiz 2

```
…
class A
{
public:
  void f()const { cout << "Arg!"; }
};

class B : public virtual A
{ };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```

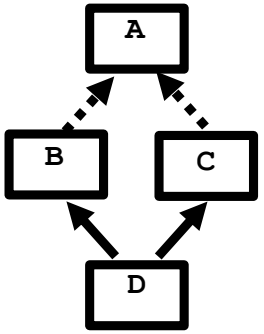


Question2 : ce code …

A ne compile pas
B compile et n'affiche rien
C compile et affiche **`Arg!`**
D compile et affiche **`Arg!Arg!`**
E compile et affiche **`Arg!Arg!Arg!`**

Héritage multiple : Quiz 3

```
…
class A
{
public:
  void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C { };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```

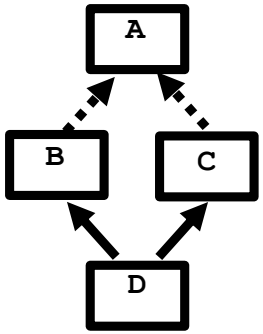


Question3 : ce code …

A ne compile pas
B compile et affiche **ABC**
C compile et affiche **AB**
D compile et affiche **AC**
E compile et affiche **BC**
F compile et affiche **A**

Exemple 4 : avec `virtual` et `using`

```
class A
{
public:
  void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{ public: using C::f; };

int main()
{
  D d1;
  d1.f();
  return 0;
}
```

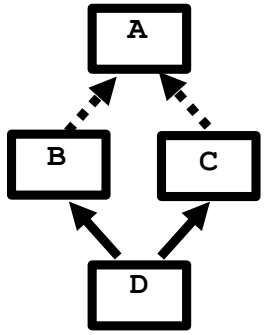


Examinons l'usage de **using** :

- La classe **D** lève l'ambiguïté sur la méthode **f** à utiliser pour ses instances en indiquant qu'il faut utiliser la méthode de la **classe C**
- C'est ce qui est fait pour l'instance d1 : l'exécution affiche C

Exemple 5 : avec `virtual` pour méthode, `virtual` pour héritage et `using`

```
class A
{
public:
  virtual void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{ public: using C::f; };

int main()
{
  return 0;
}
```



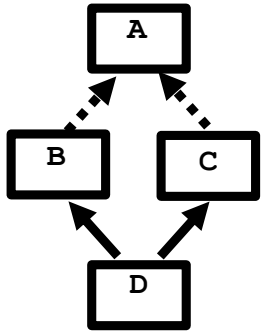
Malgré l'indication fournie par using ce code ne compile pas.

Message d'erreur: no unique final overrider

Justification: C++11 Standard 10.3/2 "In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed."

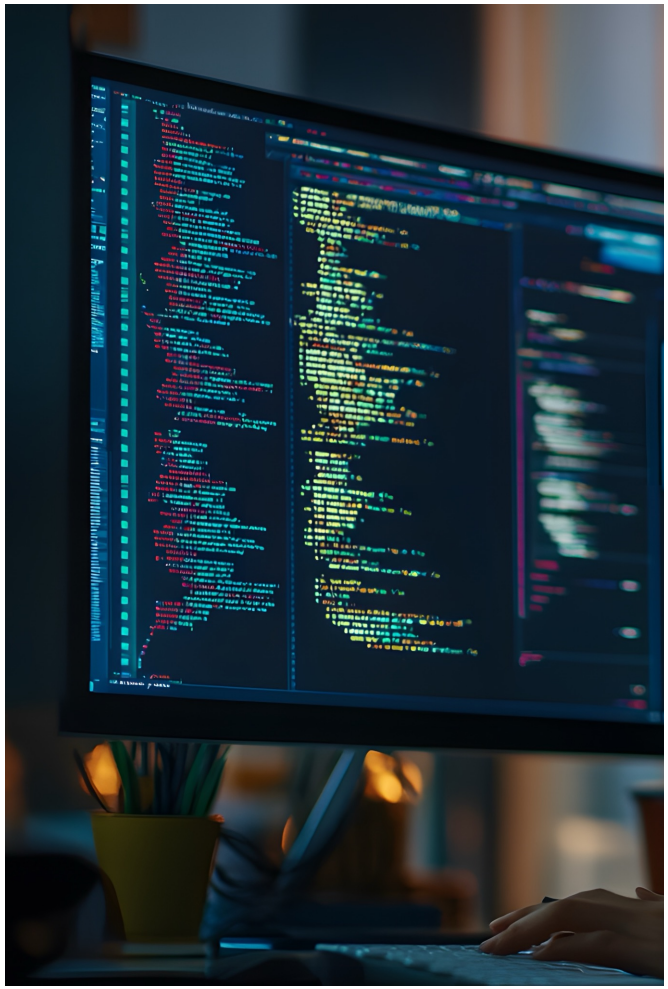
Exemple 6 : avec `virtual` pour méthode, `virtual` pour héritage

```
class A
{
public:
  virtual void f() const { cout << "A "; }
};

class B : public virtual A
{
public:
  void f() const { cout << "B "; }
};

class C : public virtual A
{
public:
  void f() const { cout << "C "; }
};

class D : public B, public C
{
public:
  void f() const { C::f(); }
};

int main()
{
  return 0;
}
```



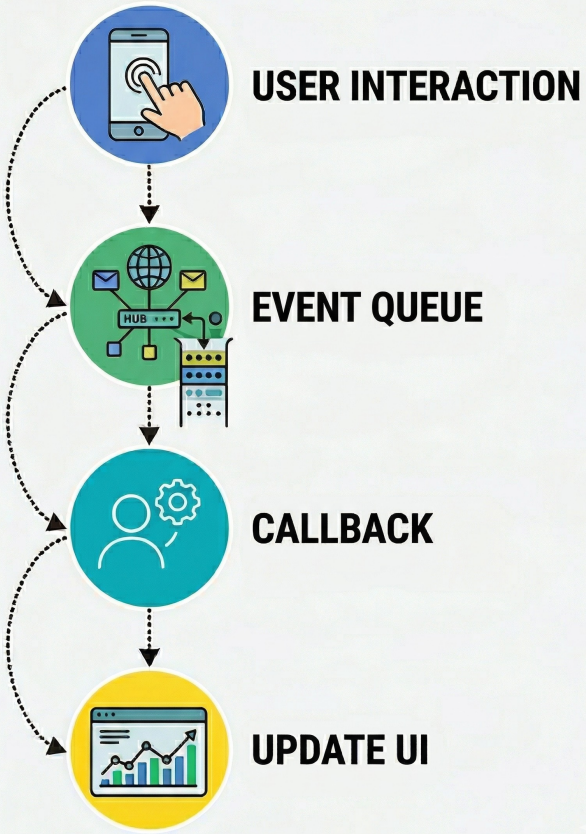
Solution: implémenter la méthode dans la classe où se trouve l'ambiguïté.

PoP 07



- **MOOC :**
  - Héritage multiple
- **Projet :**
  - Gtkmm4 : clavier et souris

Architecture d'un programme interactif graphique : clavier & souris



Objectifs:

- Usage des événements du clavier
- Usage des boutons et du mouvement de la souris

Remarques:

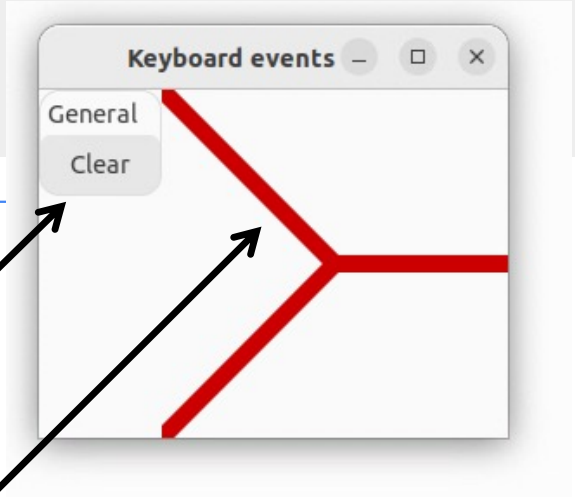
Revoir le cours de la semaine précédente sur la gestion d'événements liés à des boutons

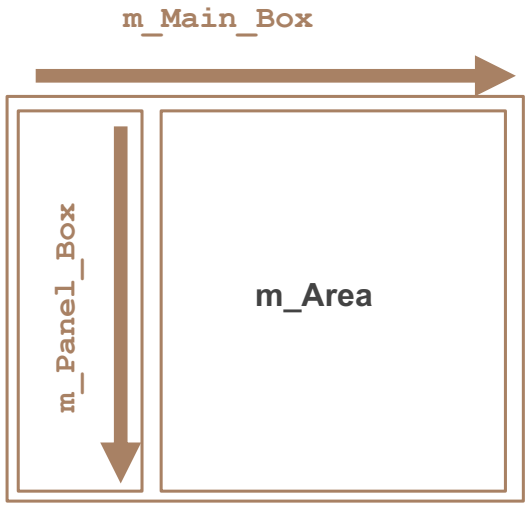
Le code présenté dans ce cours est fourni et détaillé dans la série6 niveau 0.

Exemple 1 : clavier (1)

myevent.h

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


private:
  // GUI layout
  Gtk::Box m_Main_Box;
  Gtk::Box m_Panel_Box;
  Gtk::Box m_Buttons_Box;
  Gtk::Frame m_Panel_Frame;
  Gtk::Button m_Button_Draw_Clear;
  Gtk::DrawingArea m_Area;

  bool draw ; // current drawing state
  bool thick_linewidth ; // another state variable
  //Button Signal handlers:
  void on_button_clicked_draw_clear();
  // DrawingArea signal handler:
  void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
  int width, int height);

  // keyboard event signal handler:
  bool on_window_key_pressed(guint keyval, guint keycode,
                             Gdk::ModifierType state);
};
```





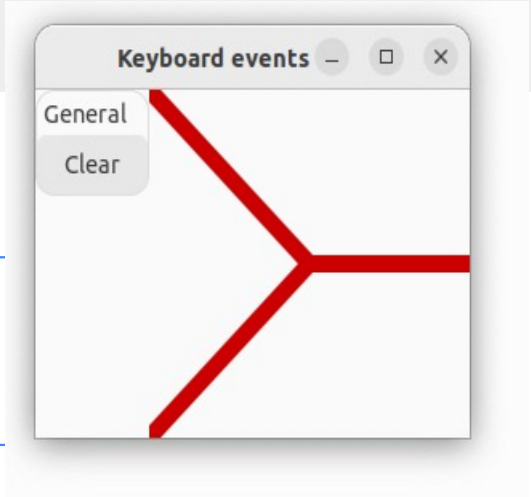
EPFL

Exemple 1 : clavier (2)

myevent.cc

à la fin du constructeur:

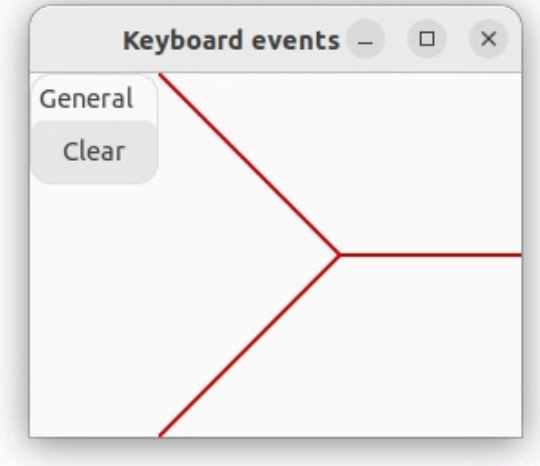
```
// Handling Keyboard Events.
auto controller = Gtk::EventControllerKey::create();
controller->signal_key_pressed().connect(
sigc::mem_fun(*this, &MyEvent::on_window_key_pressed), false);
add_controller(controller);
```




```
void MyEvent::on_button_clicked_draw_clear()
{
  std::cout << "changing drawing state through the button" << std::endl;
  draw = !draw;
  if(draw) m_Button_Draw_Clear.set_label("CLEAR!");
  else     m_Button_Draw_Clear.set_label("DRAW !");
  m_Area.queue_draw();
}
```

dans on-draw():

```
    if(thick_linewidth) cr->set_line_width(10.0);
    else cr->set_line_width(2.0);
```



EPFL

Exemple 1 : clavier (3)

```
bool MyEvent::on_window_key_pressed(guint keyval, guint, Gdk::ModifierType state)
{
  switch(gdk_keyval_to_unicode(keyval))
  {
  case 'c':
  case 'd':
    std::cout << "changing drawing state through the keyboard" << std::endl;
    draw = !draw;
    if(draw) m_Button_Draw_Clear.set_label("CLEAR!");
    else     m_Button_Draw_Clear.set_label("DRAW !");
    m_Area.queue_draw();
    return true;
  case 'w':
    std::cout << " changing state variable thick_linewidth" << std::endl;
    thick_linewidth= !thick_linewidth;
    m_Area.queue_draw();
    return true;
  case 'q':
    std::cout << "Quit" << std::endl;
    hide(); //ou exit(0);
    return true;
  }
  //the event has not been handled
  return false;
}
```

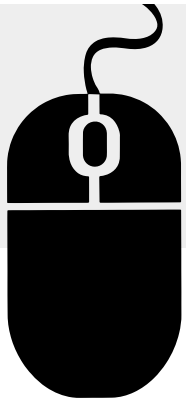
même action que le bouton m_Button_Draw_Clear

inverse l'état de la variable thick_linewidth

quitte le programme

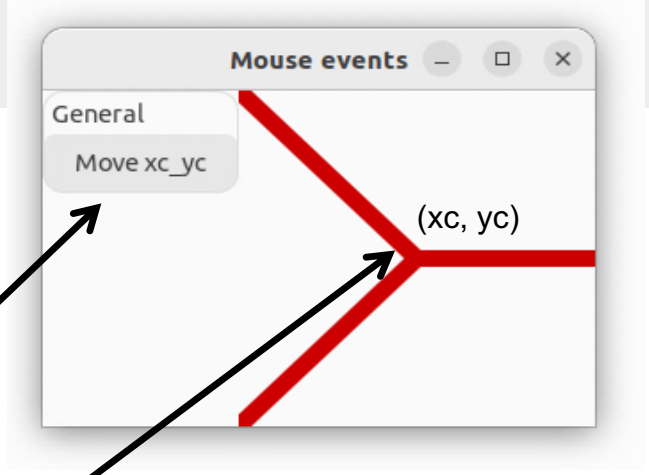
EPFL

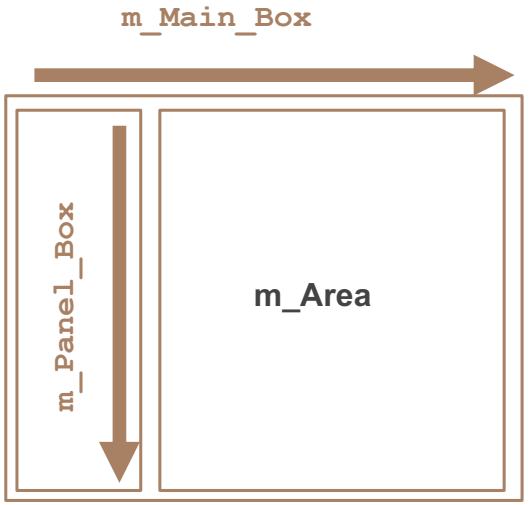
Exemple 2 : souris (1)



myevent.h

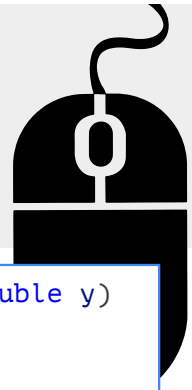
```
class MyEvent : public Gtk::Window
{
public:
  MyEvent();


private:
  // GUI layout
  Gtk::Box m_Main_Box;
  Gtk::Box m_Panel_Box;
  Gtk::Box m_Buttons_Box;
  Gtk::Frame m_Panel_Frame;
  Gtk::Button m_Button_Move_xc_yc;
  Gtk::DrawingArea m_Area;

  // used to enable using mouse move data
  bool move_xc_yc ;
  int xc,yc; // drawing parameter
  //Button Signal handlers:
  void on_button_clicked_move_xc_yc();
  // DrawingArea signal handler:
  void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
  int width, int height);

  // mouse event signal handler:
  void set_mouse_controller();
  void on_drawing_left_click(int n_press, double x, double y);
  void on_drawing_right_click(int n_press, double x, double y);
  void on_drawing_move(double x, double y);
```





EPFL

Exemple 2 : souris (2)



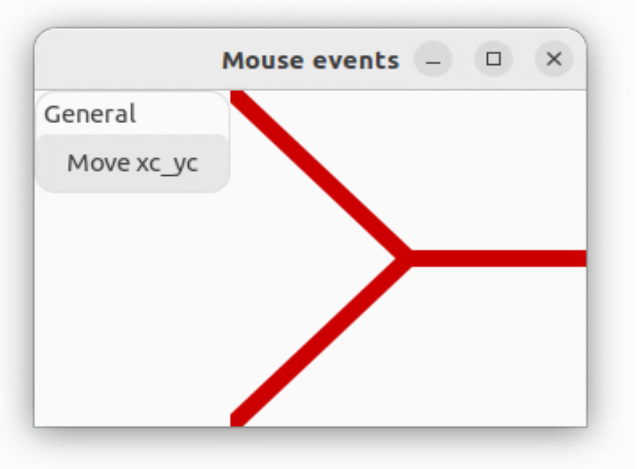
```
void MyEvent::set_mouse_controller() {
  auto left_click = Gtk::GestureClick::create();
  auto right_click = Gtk::GestureClick::create();
  auto move = Gtk::EventControllerMotion::create();

  left_click->set_button(GDK_BUTTON_PRIMARY);
  right_click->set_button(GDK_BUTTON_SECONDARY);

  left_click->signal_pressed().connect(
      sigc::mem_fun(*this, &MyEvent::on_drawing_left_click));
  right_click->signal_pressed().connect(
      sigc::mem_fun(*this, &MyEvent::on_drawing_right_click));
  move->signal_motion().connect(
      sigc::mem_fun(*this, &MyEvent::on_drawing_move));

  m_Area.add_controller(left_click);
  m_Area.add_controller(right_click);
  m_Area.add_controller(move);
}
```

```
void MyEvent::on_drawing_right_click(int n_press, double x, double y)
{
  cout << "mouse right clic detected and processed" << endl;
  move_xc_yc = !move_xc_yc;
  if(move_xc_yc ) m_Button_Move_xc_yc.set_label("Do Not Move!");
  else            m_Button_Move_xc_yc.set_label("Move (xc,yc)");
  m_Area.queue_draw();
}
```

inverse l'état de la variable move_xc_yc

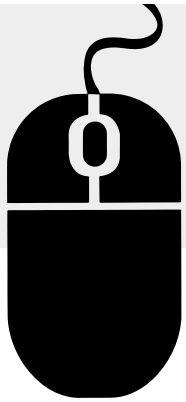
myevent.cc

```
void MyEvent::on_drawing_left_click(int n_press, double x, double y)
{
  cout << "mouse left clic detected and processed" << endl;
  cout << "x = " << x << " y = " << y << endl;
  xc = x;
  yc = y;
  m_Area.queue_draw();
}
```



EPFL

Exemple 2 : souris (3)



```
void MyEvent::on_drawing_move(double x, double y)
{
  static char c('A');
  cout << c << " => mouse move detected and processed" << endl;
  if(++c >'Z') c= 'A'; // reset

  if(move_xc_yc)
  {
    if(x > 0 and x < m_Area.get_width())
      xc = x;
    if(y > 0 and x < m_Area.get_height())
      yc = y;
  }
  else
  {
    xc = m_Area.get_width()/2 ;
    yc = m_Area.get_height()/2 ;
  }
  m_Area.queue_draw();
}
```

```
A => mouse move detected and processed
B => mouse move detected and processed
C => mouse move detected and processed
D => mouse move detected and processed
E => mouse move detected and processed
F => mouse move detected and processed
G => mouse move detected and processed
H => mouse move detected and processed
mouse left clic detected and processed
x = 73   y = 96
mouse right clic detected and processed
I => mouse move detected and processed
J => mouse move detected and processed
K => mouse move detected and processed
```

lancer l'exécutable pour constater l'effet sur le dessin

Résumé

- Comme pour les autres éléments d'une interface graphique, la clef de leur fonctionnement est l'usage d'attributs pour mémoriser un *état désiré* de l'application
- ne pas oublier d'appeler `queue_draw()` si le changement désiré a un impact visuel.
- les événements de mouvement de la souris sonts produit seulement si un mouvement minimum est effectué.

rafael.pires@epfl.ch

Merci