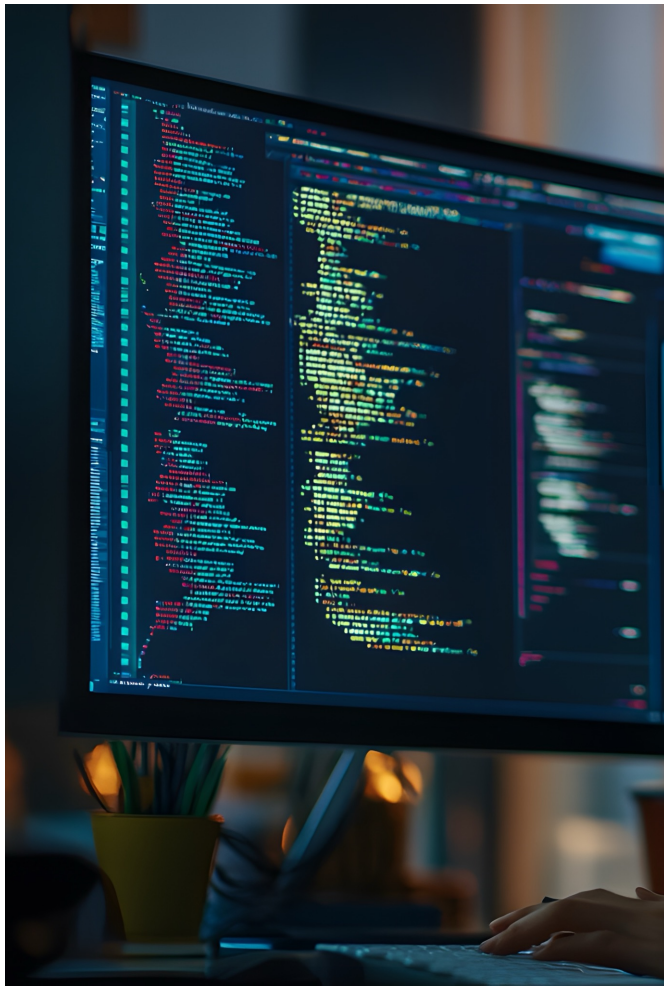
Programmation Orientée Projet
COM-112(a)

Semaine 5

Rafael Pires
rafael.pires@epfl.ch

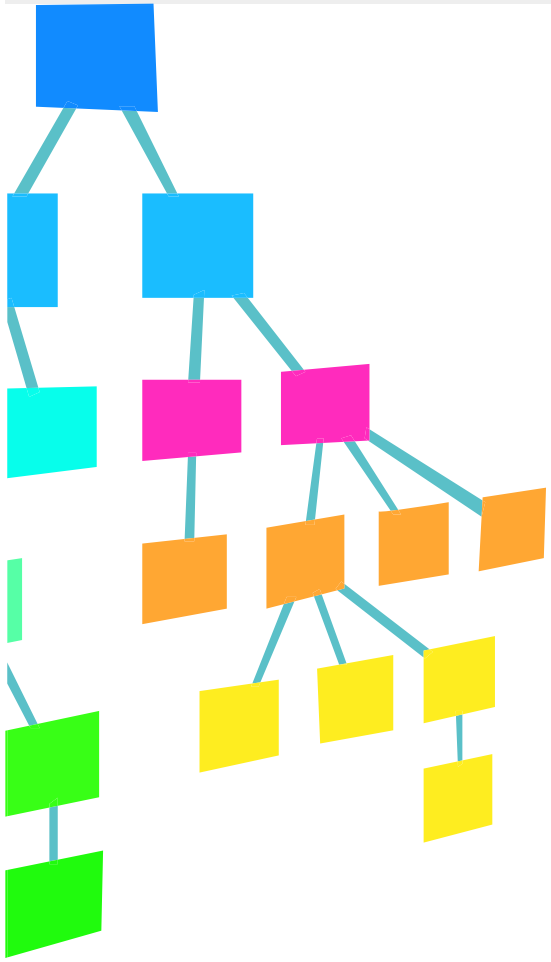
Lausanne, 20.03.2026

PoP 05



- MOOC :
  - Héritage
- Projet :
  - Pointeur de fonction
  - L'architecture MVC
  - Gtkmm4

Héritage : Distinguer « est un(e) » de « possède un(e) »

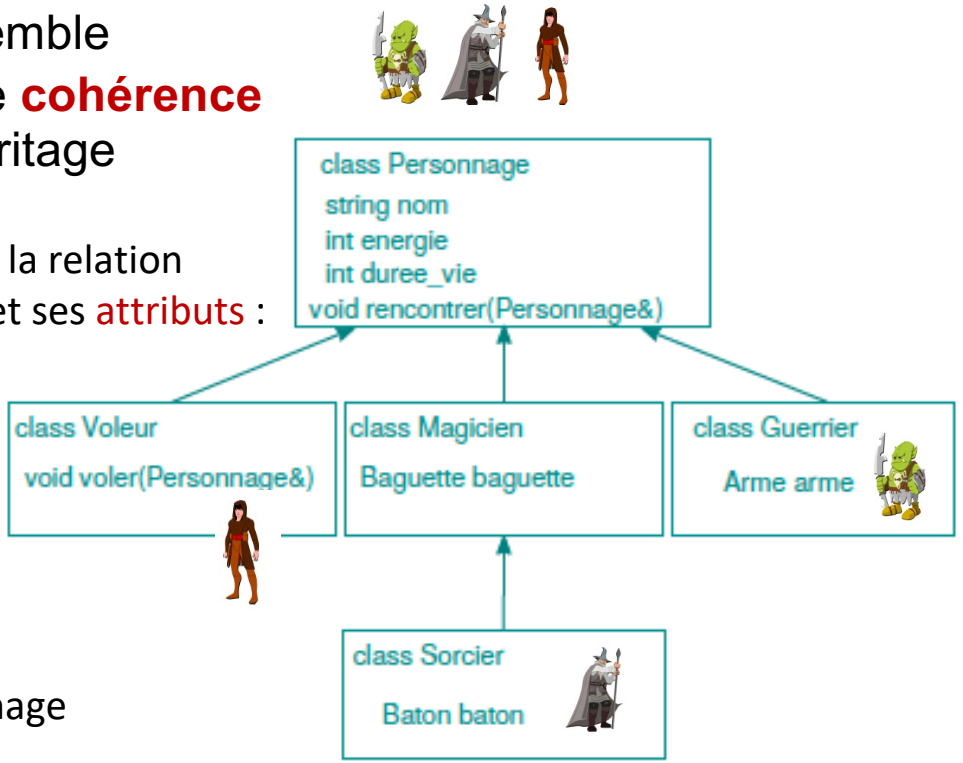


- **Motivation :** lorsqu'un ensemble d'entités présente une forte **cohérence sémantique**, on utilise l'héritage

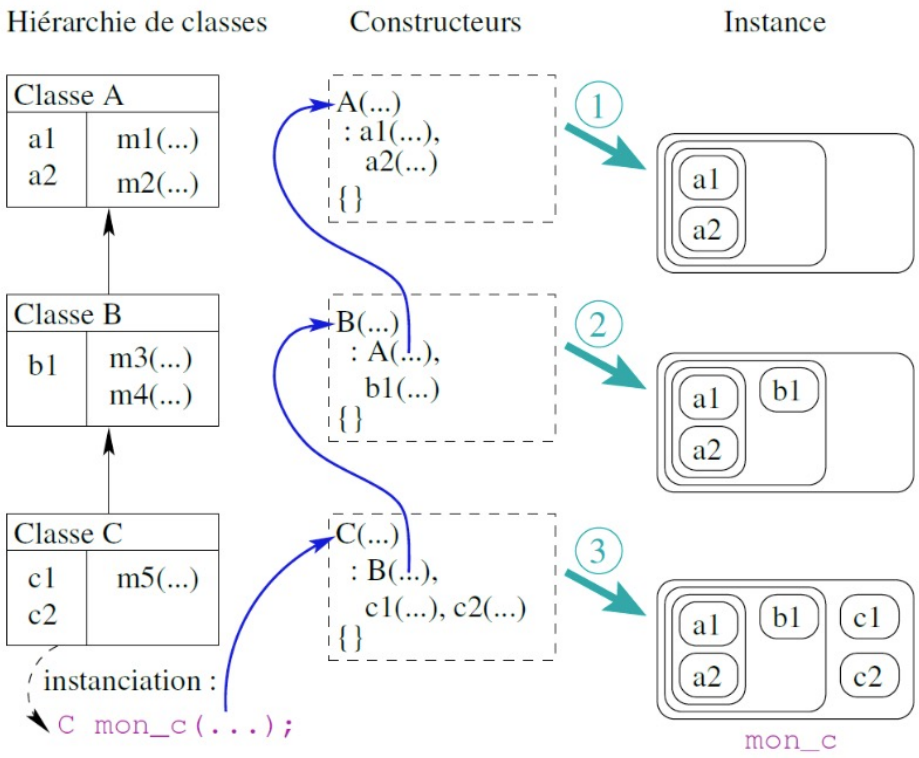
❖ La relation **« possède un(e) »** est la relation entre une instance d'une classe et ses attributs : Un personnage possède : `nom`, `energie`, `duree_vie`

❖ La relation **« est un(e) »** est celle qui existe entre une sous-classe et sa classe parente :

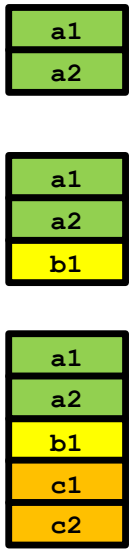
- Un Sorcier est un Magicien
- Un Magicien est un Personnage



Héritage : Attributs de la hiérarchie de classes



En mémoire

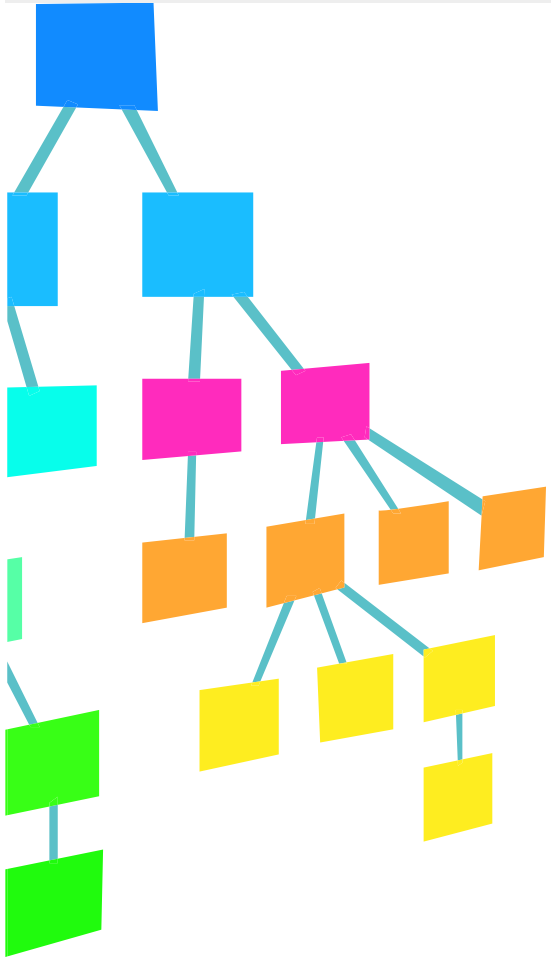


Conséquence (slicing)

- L'affectation d'une instance d'une classe dérivée à une instance de sa classe parente garde **seulement** les attributs de la classe parente.
- La «tranche» (slice) des attributs de la classe dérivée **est perdue**

```
A x;
B y;
C z;

// autorisé mais perte
x = y;
y = z;
x = z ;
```

Quiz : Surcharge & masquage



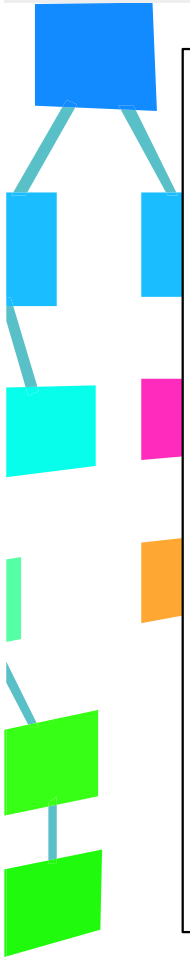
prog.cc

```
1  #include <iostream>
2  #include <string>
3  class A {
4  public:
5      void m1(int i)         {std::cout << "A::m1(int)"    << i << std::endl;}
6      void m1(std::string s) {std::cout << "A::m1(string)" << s << std::endl;}
7  };
8  class B: public A {
9  public:
10     void m1(std::string s) {std::cout << "B::m1(string)" << s << std::endl;}
11     void h(int i)          { m1(i); }
12 };
```

La compilation (option –c) du fichier source `prog.cc` produit...

A un fichier **prog.o** sans aucun warning
B une erreur à la ligne 10 car **m1** est déjà définie par la classe A
C une erreur à la ligne 11 à cause de l'appel de **m1**
D une erreur pour une autre raison

Quiz : Hiérarchie de classes



prog.cc

```
#include <iostream>
class A {
protected:
    int x;
};

class B: public A {
protected:
    int x1;
};

class C: public B {
public:
    void h(A *p1, B* p2, C* p3);
private:
    int x2;
};

void C::h(A *p1, B* p2, C* p3){
    C *p4(new C);

    std::cin >> p1->x;        // 1
    std::cin >> p2->x1;       // 2
    std::cin >> p3->x2;       // 3
    std::cin >> p4->x;        // 4
}
```

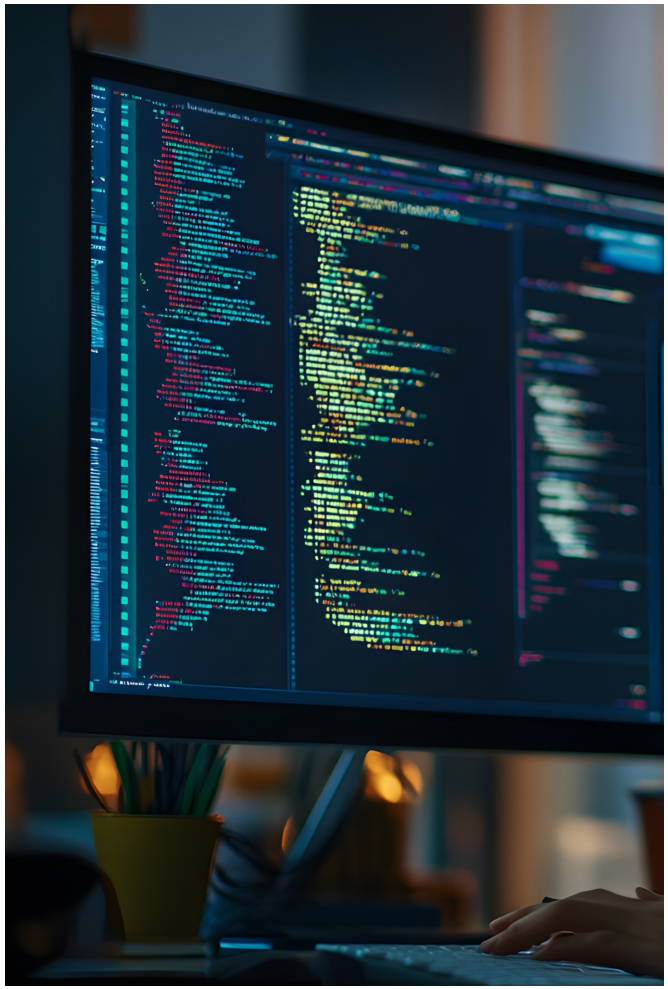
On compile (option –c) le fichier source prog.cc déclarant la hiérarchie de classes A-B-C.

On s'intéresse aux 4 lignes de lecture sur std::cin avec des commentaires numérotés de 1 à 4.

Chacune de ces lignes compile-t-elle correctement ?
Réponses proposées (oui/non) dans l'ordre des 4 lignes

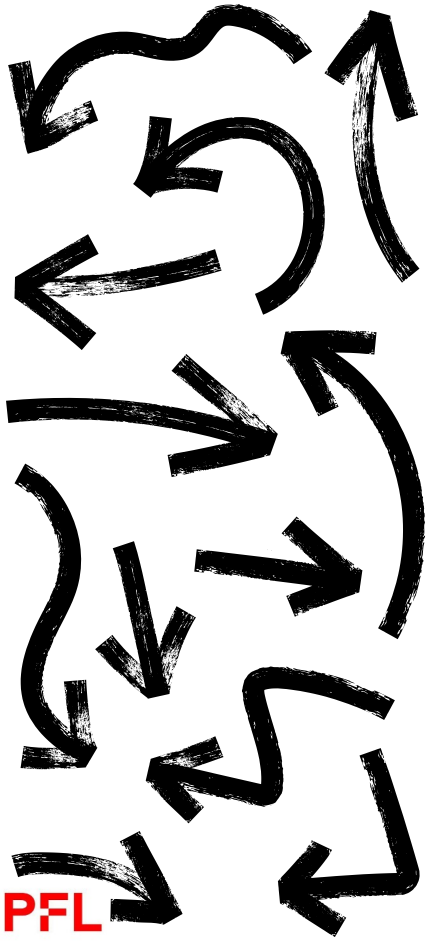
	// 1	// 2	// 3	// 4
A	oui	oui	oui	oui
B	non	oui	oui	non
C	non	non	oui	non
D	non	non	oui	oui

PoP 05



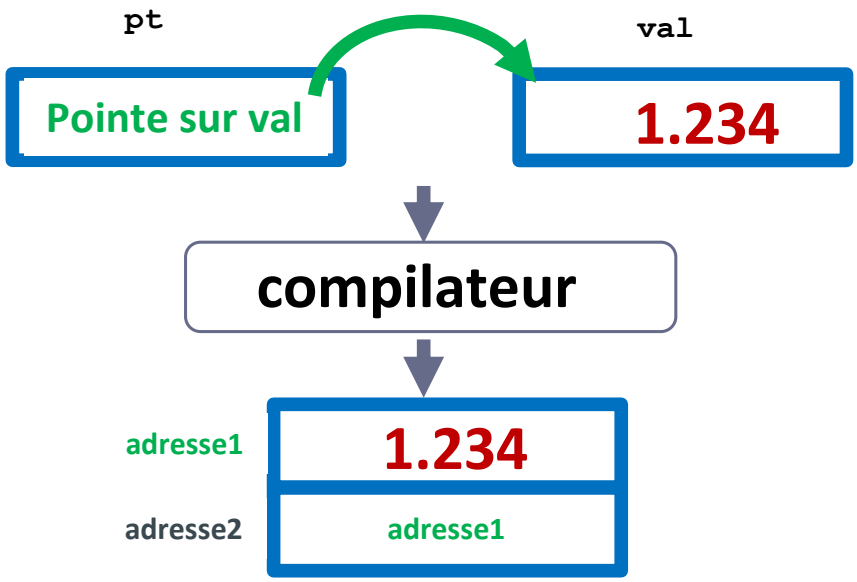
- MOOC :
  - Héritage
- Projet :
  - Pointeur de fonction
  - L’architecture MVC
  - Gtkmm4

Rappel : Pointeurs



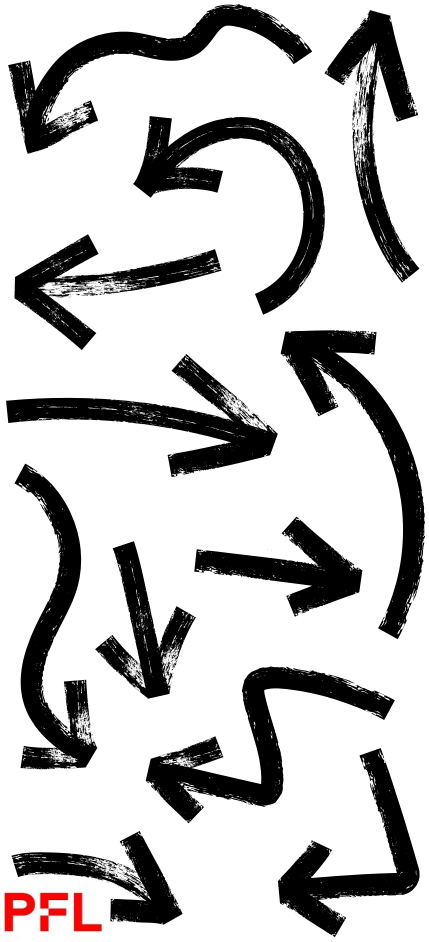
Un pointeur est une **variable** indépendante qui mémorise une **adresse** d'un octet de la mémoire

```
double val(1.234);
double *pt = &val;
```



adresse1 est l'adresse du premier octet occupé par val

Pointeur sur une fonction



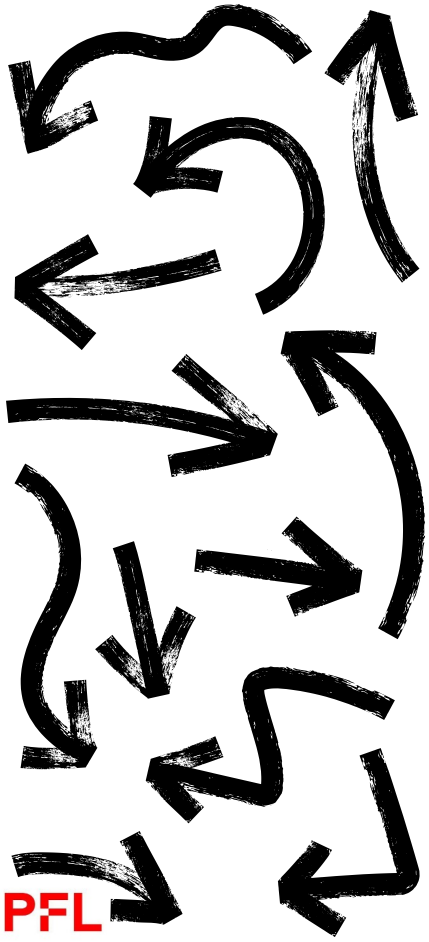
- une fonction peut être « passée en paramètre » à une autre fonction simplement en transmettant son nom. Car le **nom de la fonction**, sans les parenthèses, est l'**adresse** de sa première instruction exécutable
  - l'argument formel correspondant est de type **pointeur de fonction**

    dont le prototype doit correspondre à celui de la fonction transmise

Ex: soit le prototype `int f(double,int);`

`f(2.5, 9)`	applique l'opérateur appel de fonction `( )` sur `f`
`&f`	est l'adresse de la fonction `f`
`f`	désigne AUSSI l'adresse de cette fonction `f`

Pointeur sur une fonction



- Le type d'un ***pointeur de fonction*** doit faire apparaitre:
  - le fait qu'il mémorise une adresse
  - le prototype de la fonction pointée

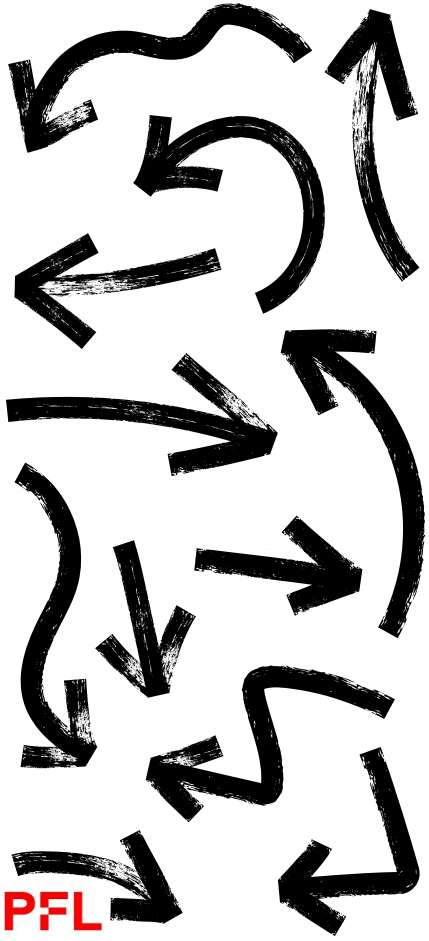
- Soit le prototype `int f(double,int);`
  On déclare un pointeur **adf** sur une fonction de ce prototype :

```
int (*adf)(double,int);
```

- Les parenthèses autour de `(*adf)` sont nécessaires pour **forcer la priorité** de `*` qui déclare `adf` comme pointeur
- L'instruction suivante **ne déclare pas** `g` comme un pointeur de fonction:

```
int *g(double,int);
```

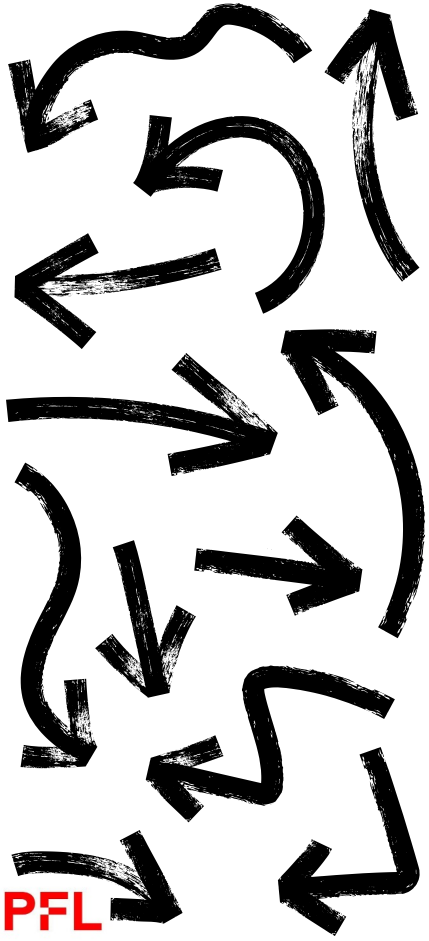
Initialisation et usage d'un pointeur de fonction



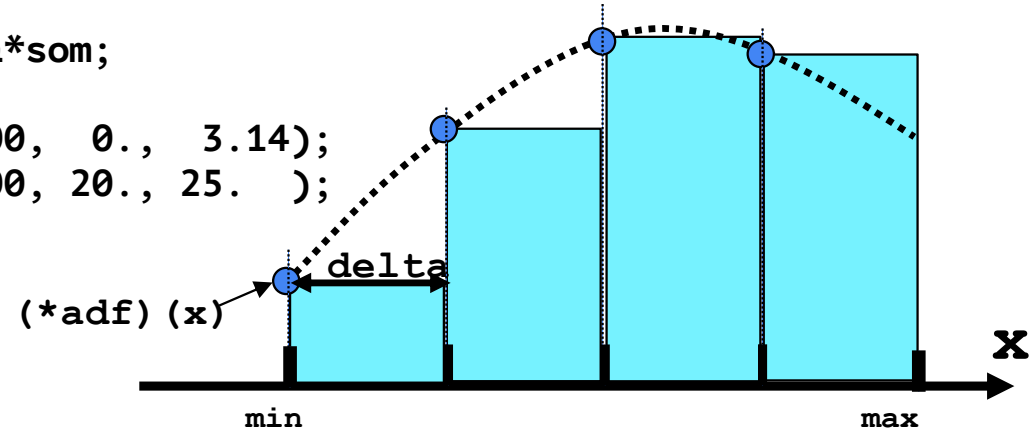
```
1    int  f(double, int);
2          …
3    f(12.3, 5);
4          …
5   int (*adf)(double, int); // n'est pas initialisé
6     …
7   adf = f;                 // ou: adf = &f;
8     …
9   (*adf)(0.5, 2);          // appelle la fonction f
10  adf(0.5, 2);             // aussi correct en C++
11  f(0.5,2);
```

Les lignes 9 et 10 sont possibles en C++ pour appeler la fonction à l'aide de **`adf`**
C'est équivalent à appeler directement f comme sur la ligne 11

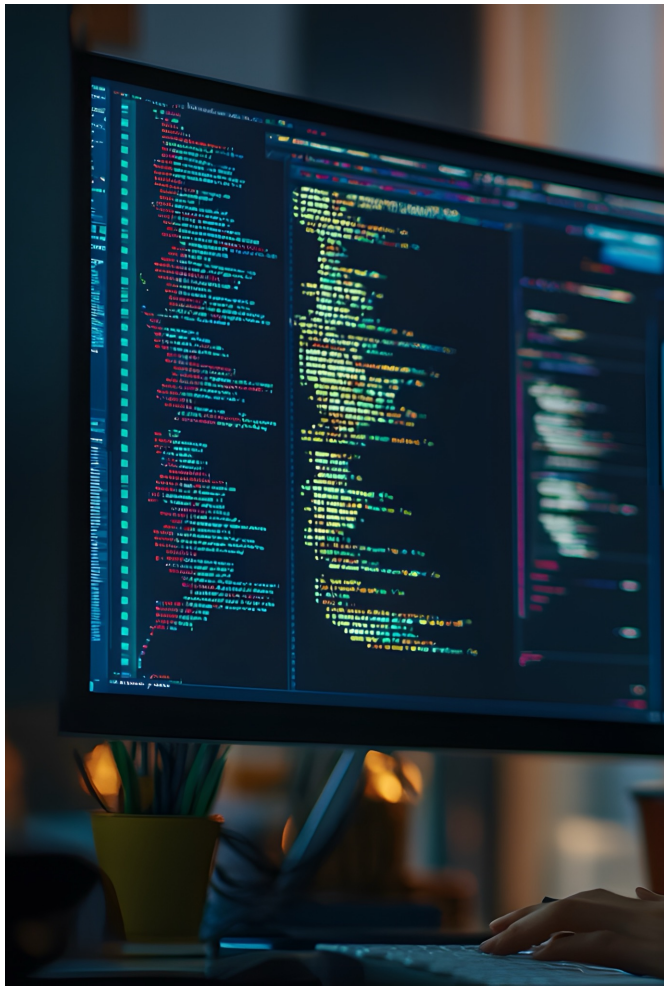
Passage d'une fonction en paramètre d'une fonction



```
1    double  sin(double);
2    double  sqrt(double);
          …
3    double integ( double(*adf)(double), int nb,
4                  double min, double max)
5    {
6        double delta((max-min)/nb), x(min), som(0);
7
8        for(int i(0); i<nb ; ++i, x += delta)
9           som += (*adf)(x) ;  // aussi ok avec adf(x)
10
11      return delta*som;
12   }
13   integ(sin,   100,  0.,  3.14);
14   integ(sqrt, 1000, 20., 25.  );
```



PoP 05



- **MOOC :**
  - Héritage
- **Projet :**
  - Pointeur de fonction
  - **L’architecture MVC**
  - **Gtkmm4**

Les 3 styles de dialogues humain-machine

- **ligne de commande** (commandes UNIX-LINUX)
  - passage de paramètres au programme avec `argc` et `argv`
- **Conversationnel** (programmes semestre d'automne)
  - `cout`, `cin`, etc disponible avec la bibliothèque standard C++
- **Interface Graphique Utilisateur**
  (*Graphic User Interface* / **GUI**)
  - widgets (boutons, etc...) disponible dans ***bibliothèques***

Bibliothèque (*Library*) : regroupe le code objet de plusieurs modules réalisant une tâche bien définie (exemple: interface graphique, dessin, etc...)

API (*Application Programming Interface*) : prototypes des fonctions exportées par une bibliothèque dans un fichier en-tête (exemple: **gtkmm.h**) = fichier d'interface de la bibliothèque

L'architecture MVC : Model-View-Controller

Principe : séparation des responsabilités

Objectifs:

- Permettre plusieurs styles de Visualisations et de Contrôle/Dialogue sur un Modèle.
- S'adapter facilement à l'évolution rapide de la technologie des parties "Visualisation" et "Contrôle utilisateur" en gardant stable la partie "Modèle métier"

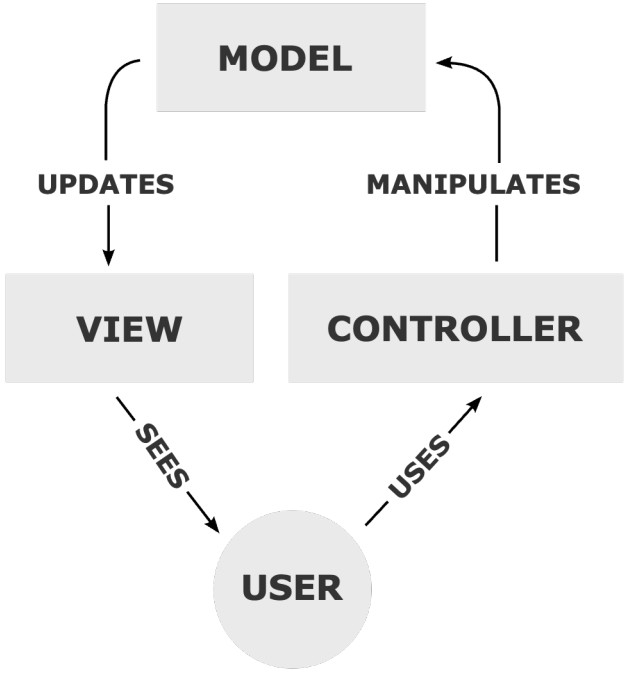
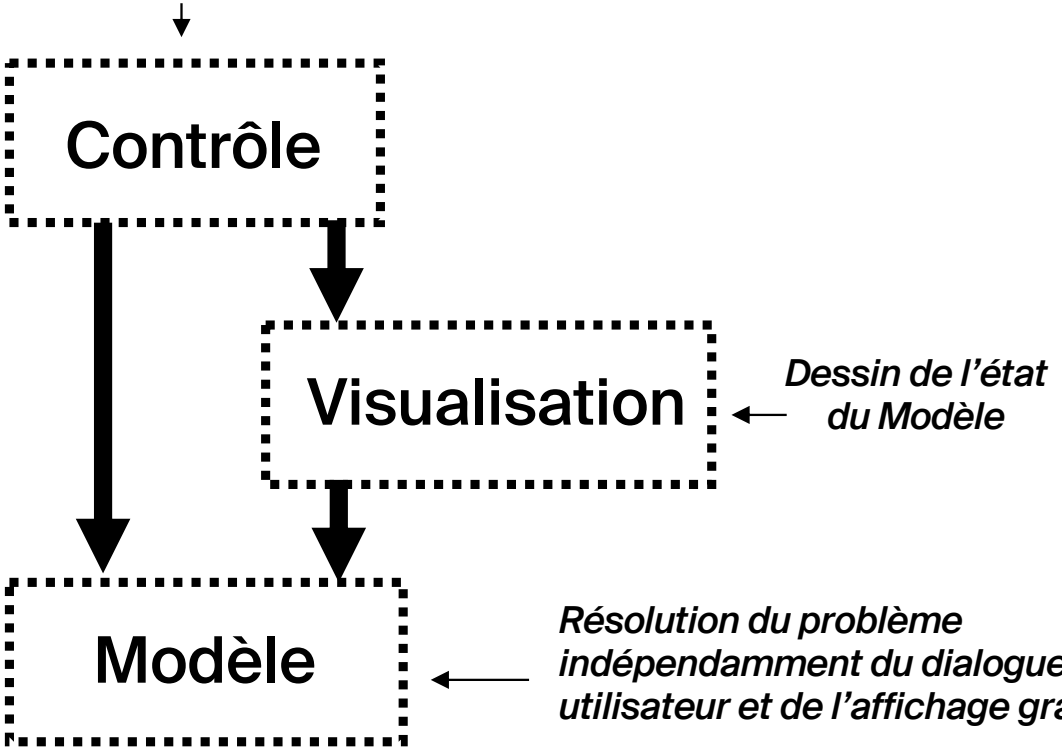


Diagramme conceptuel
[wikipedia]

L'architecture MVC : Model-View-Controller

Dialogue humain-machine:
Ligne de commande, conversationnel ou GUI



Dessin de l'état du Modèle

Résolution du problème indépendamment du dialogue utilisateur et de l'affichage graphique

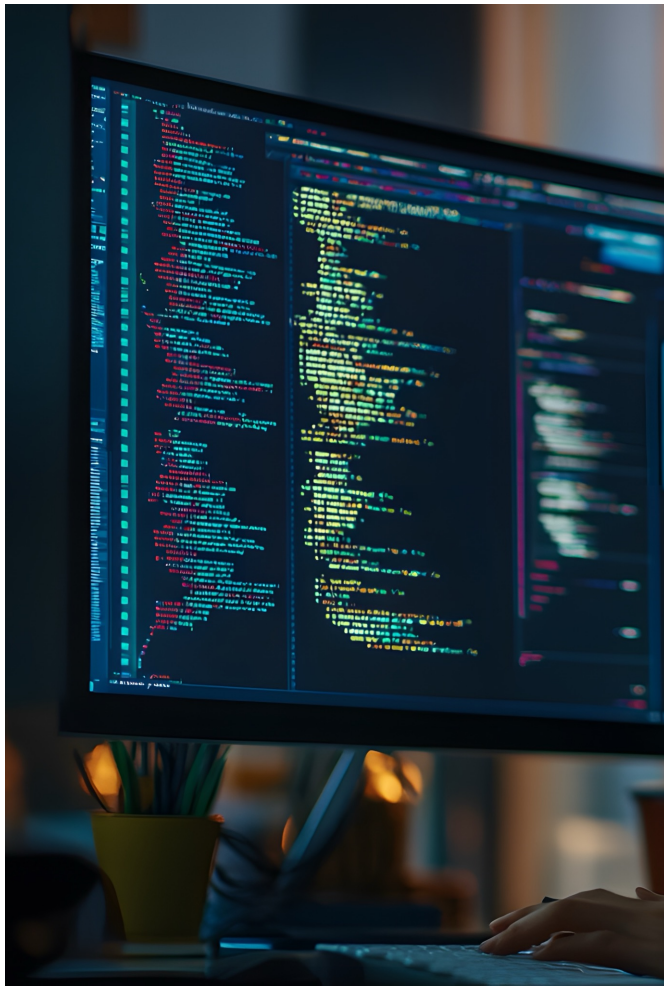
Architecture logicielle

L'architecture MVC : Résumé

- En vertu du principe de séparation des responsabilités/fonctionalités nous adoptons l'approche Model-View-Controller pour l'architecture d'une application interactive:
  - Le sous-système de Contrôle pilote celui de Visualisation et celui du Modèle du problème traité.
  - Le sous-système du Modèle n'a aucune dépendance vis à vis du dialogue humain-machine ni des fonctions de dessins

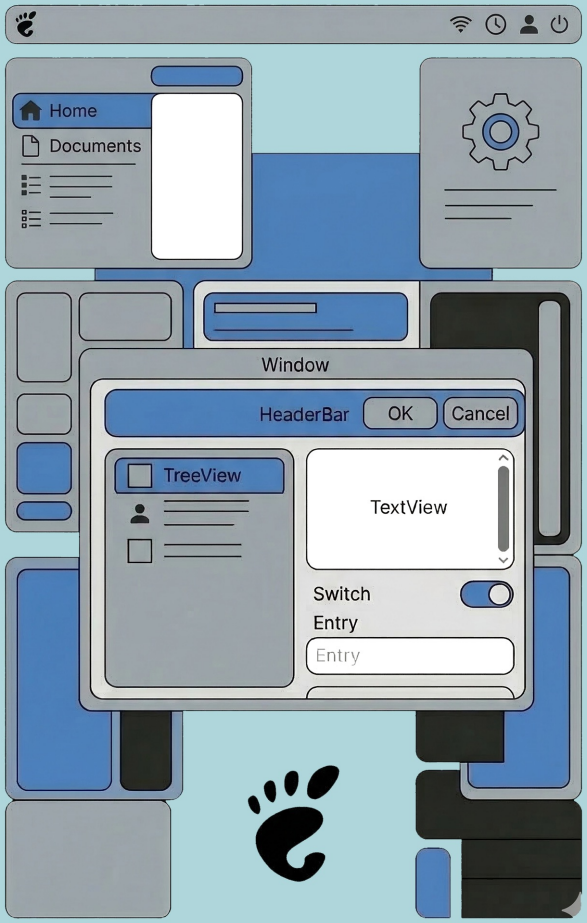
- GTKmm offre une hiérarchie de classes C++ pour construire une interface utilisateur

PoP 05



- **MOOC :**
  - Héritage
- **Projet :**
  - Pointeur de fonction
  - L'architecture MVC
  - **Gtkmm4**

GTKmm4 : Bases et visualisation

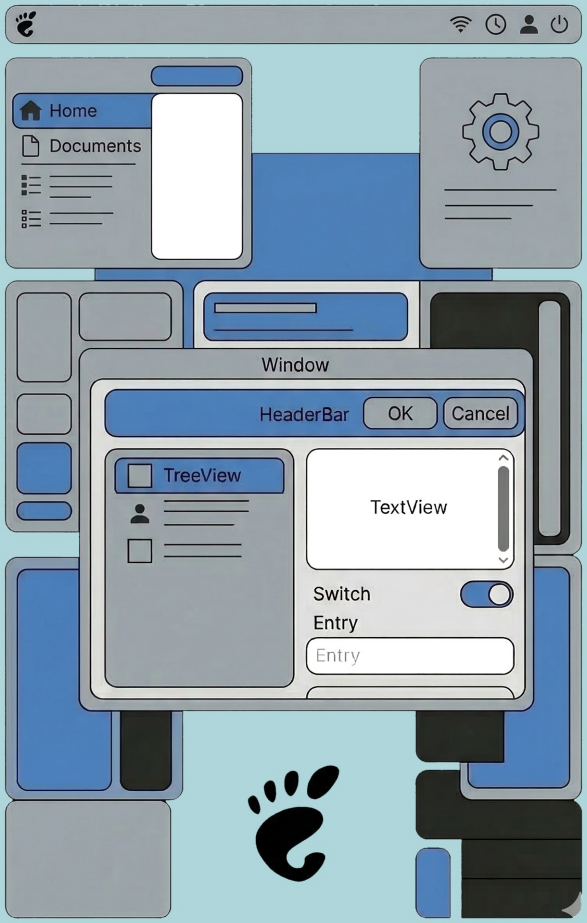


La série4 niveau0 développe les détails techniques et fournit l'ensemble du code source

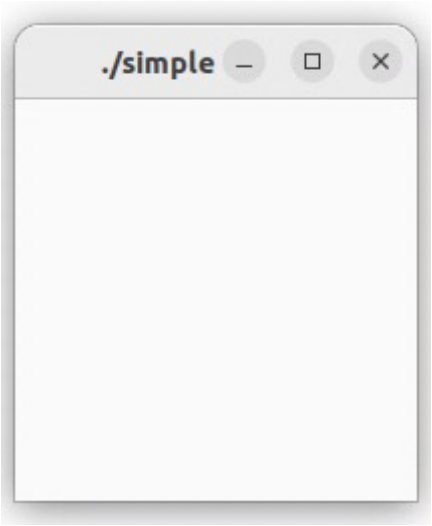
Plan:

- Première application GTKmm4
- Un premier bouton
- l'API Cairomm pour le dessin
- Le premier dessin avec MyDrawingArea

GTKmm4 : Première Application GTKmm4



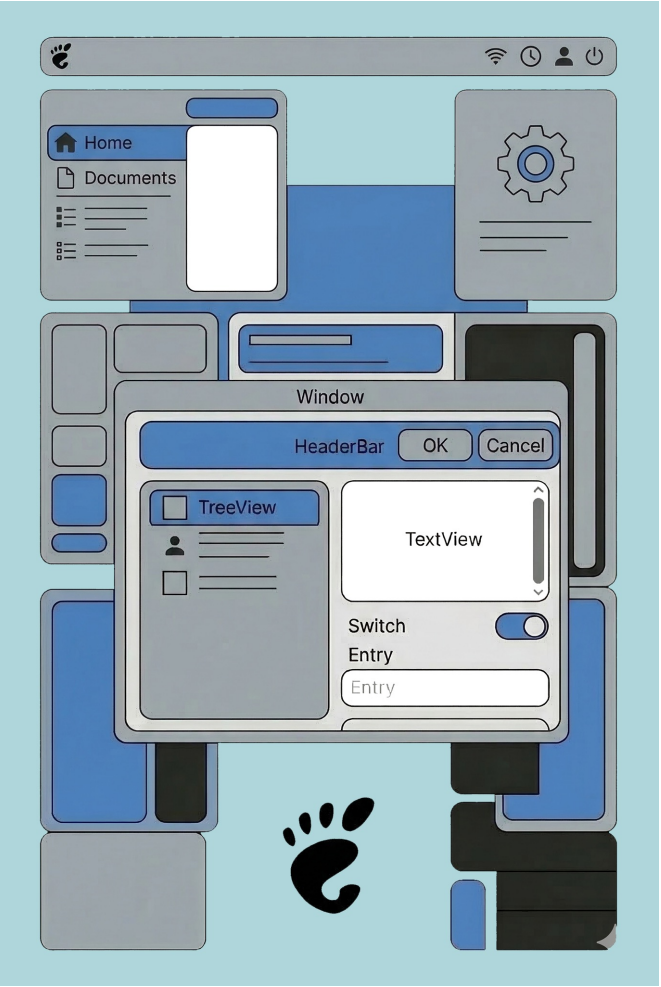
Le texte indiqué avec `set_title` *apparaît dans le bandeau supérieur.*




```
class MyWindow : public Gtk::Window {
public:
    MyWindow(std::string name);
};

MyWindow::MyWindow(std::string name) {
    set_title(name);
}


int main(int argc, char* argv[]) {
    std::string init(argc>1 ? argv[1] : argv[0]);

    auto app = Gtk::Application::create();

    return app->make_window_and_run<MyWindow>(1, argv, init);
}
```

*transfert du contrôle du programme à GTKmm avec l'appel d'une méthode **template** C++ dont le code est adapté au type <MyWindow> : création et lancement d'une fenêtre MyWindow*

GTKmm4 : Fonctions en paramètre

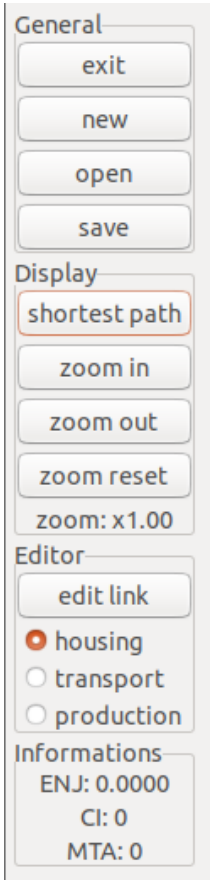


- Une bibliothèque telle que GTKmm est une hiérarchie de classes qui sert à créer une interface graphique composée de **widgets**, par exemple des ***buttons***.

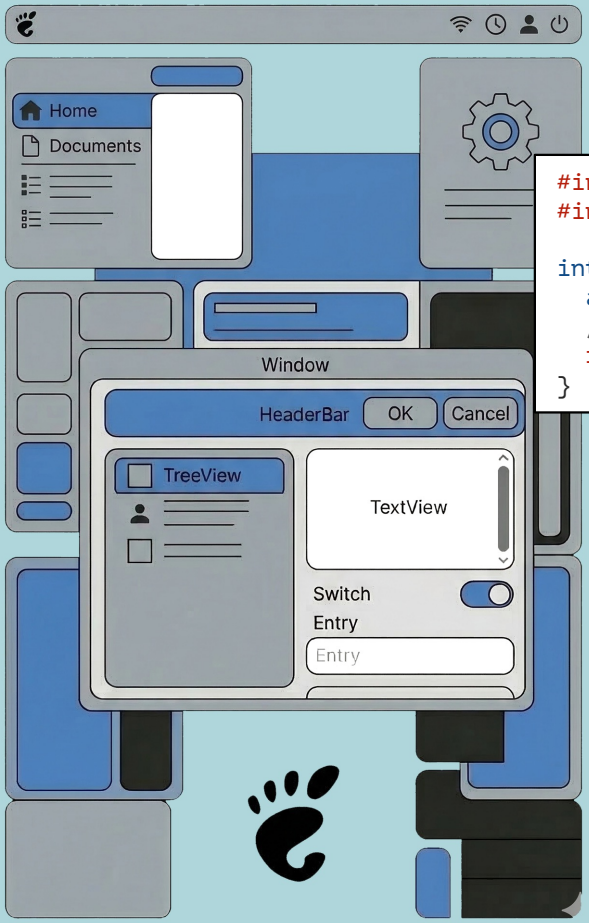
- Avec GTKmm Il est possible d'associer une **réaction**, *sous forme d'une fonction*, au fait de cliquer la souris (ou un autre **événement**) sur un ***button*** en particulier.

1. Dans une **phase d'initialisation**, pour chaque ***button*** il suffit de *passer un nom de fonction* à une méthode dédiée.
   - ➢ le **pointeur de fonction est mémorisé** dans un attribut de ***button*** dans GTKmm

2. Après l'initialisation, pendant la **boucle infinie d'interaction**, chaque fois qu'on clique sur le ***button*** la fonction associée au ***button*** est appelée à l'aide du pointeur de fonction mémorisé dans l'étape 1.



GTKmm4 : Premier bouton

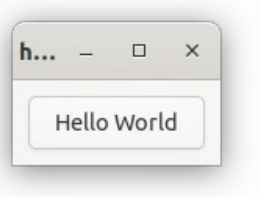


main.cc

```cpp
#include "helloworld.h"
#include <gtkmm/application.h>

int main (int argc, char *argv[]) {
  auto app = Gtk::Application::create();
  //Shows the window and returns when it is closed.
  return app->make_window_and_run<HelloWorld>(argc,argv);
}
```

helloworld.h

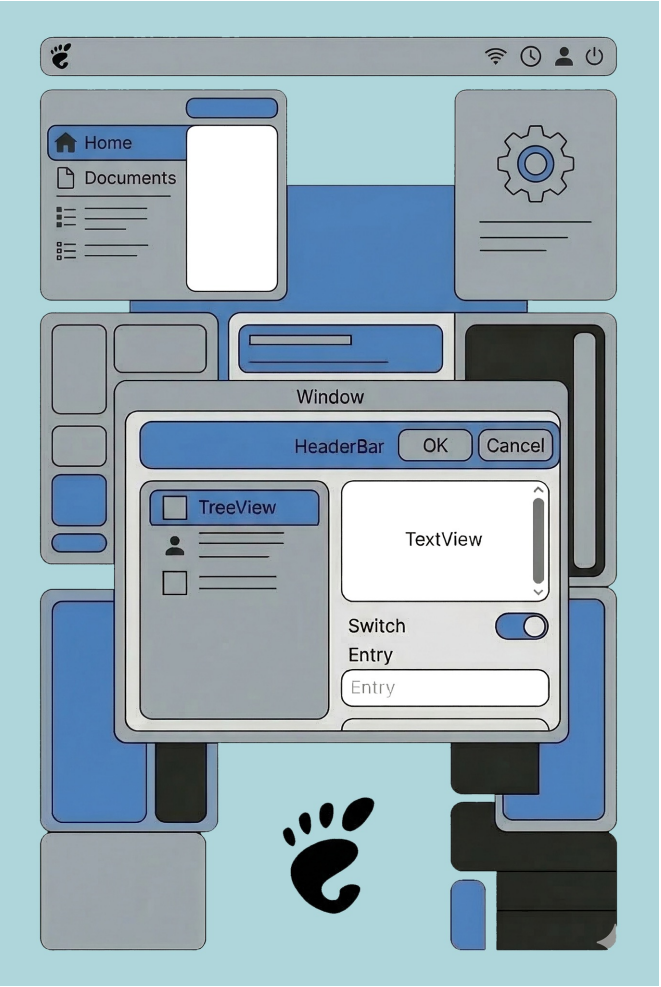
```cpp
#include <gtkmm/button.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window {
public:
  HelloWorld();
  ~HelloWorld() override;

protected:
  //Signal handlers:
  void on_button_clicked();
  //Member widget:
  Gtk::Button m_button;
};
```



helloworld.cc

```cpp
#include "helloworld.h"
#include <iostream>


HelloWorld::HelloWorld()
    : m_button("Hello World") {
  m_button.set_margin(10);

  m_button.signal_clicked()
          .connect(sigc::mem_fun(*this,
          &HelloWorld::on_button_clicked));
  set_child(m_button);// ajoute à la fenètre
}


HelloWorld::~HelloWorld(){}


void HelloWorld::on_button_clicked() {
  static unsigned count(0);
  std::cout << ++count << " Hello World \a"
            << std::endl;
}
```

GTKmm4 : Elements principaux de l'API Cairomm pour le dessin

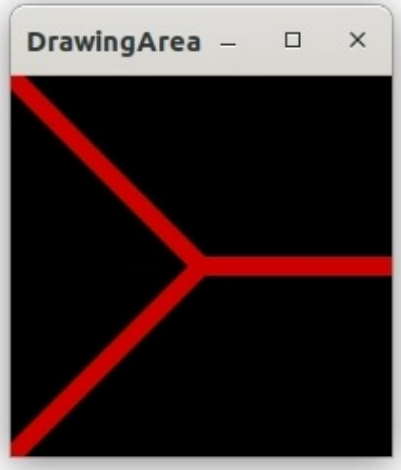


- Le widget spécialé **DrawingArea** est celui destiné au dessin

- Pour dessiner il faut définir la méthode `on_draw()`

Elle est aussi appelée automatiquement quand le système détecte que la fenêtre a besoin d'être redessinée.

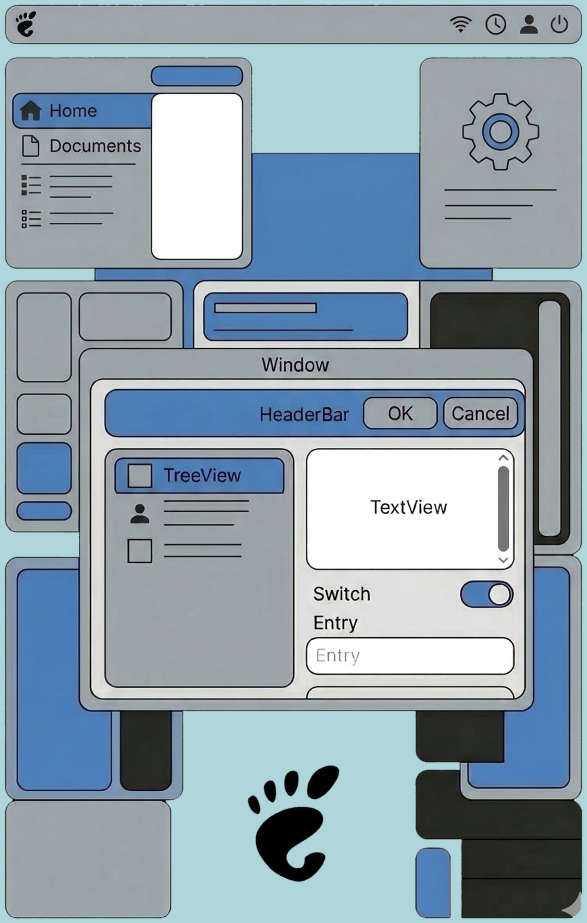
- On dispose de méthodes pour dessiner des lignes droites, courbes ou des arcs de cercle,etc...



Un objet `Cairo::Context` permet de mémoriser tous les paramètres courants du dessin tels que l'épaisseur du trait, la couleur, etc...

Ainsi les fonctions de dessin n'ont pas à préciser ces paramètres à chaque appel

GTKmm4 : Les principales commandes d'un contexte



Définir un nouvel état permanent d'un paramètre

Variable d'état	Val par défaut	Méthode pour modifier l'état
Épaisseur du trait	1	**`set_line_width`**(val)
couleur du dessin	noir	**`set_source_rgb`**(R,G,B)

Dessiner un fond de couleur uniforme avec **`paint()`**

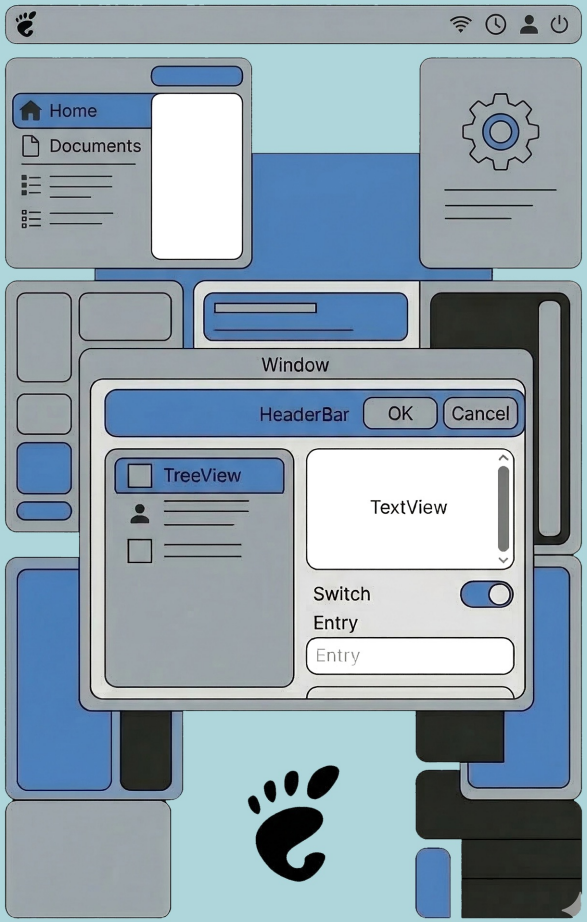
```
cr->set_source_rgb(0.0, 0.0, 0.0);
cr->paint();
```

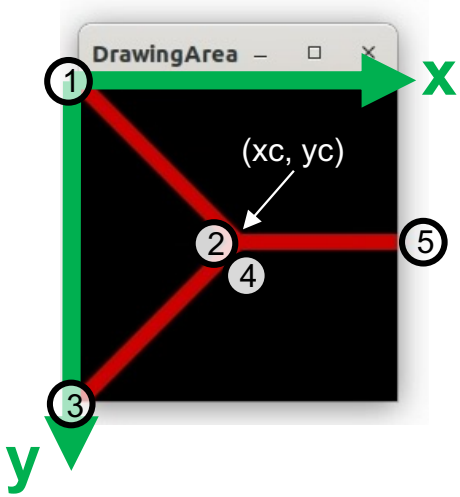
Créer une ligne ou polyline / concept de « **path** » :

Définir le début de ligne avec **`move_to(x,y)`**
Définir un point d'un path avec **`line_to(x,y)`**

Dessiner le « **path** » qui vient d'être créé: **`stroke()`**

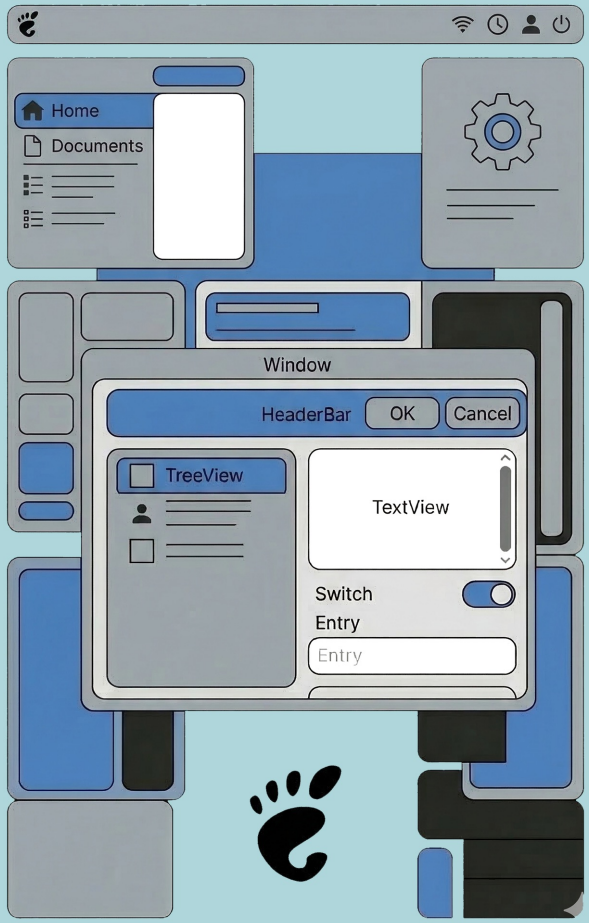
GTKmm4 : Les principales commandes d'un contexte



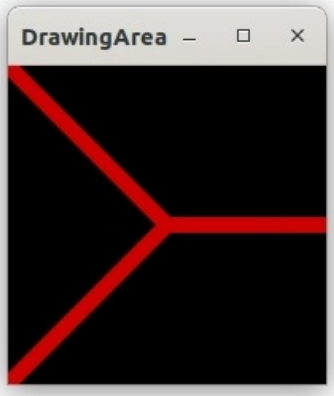



```
cr->set_line_width(10.0);
cr->set_source_rgb(0.8, 0.0, 0.0);
1 cr->move_to(0, 0);
2 cr->line_to(xc, yc);
3 cr->line_to(0, height);
4 cr->move_to(xc, yc);
5 cr->line_to(width, yc);
cr->stroke();
```

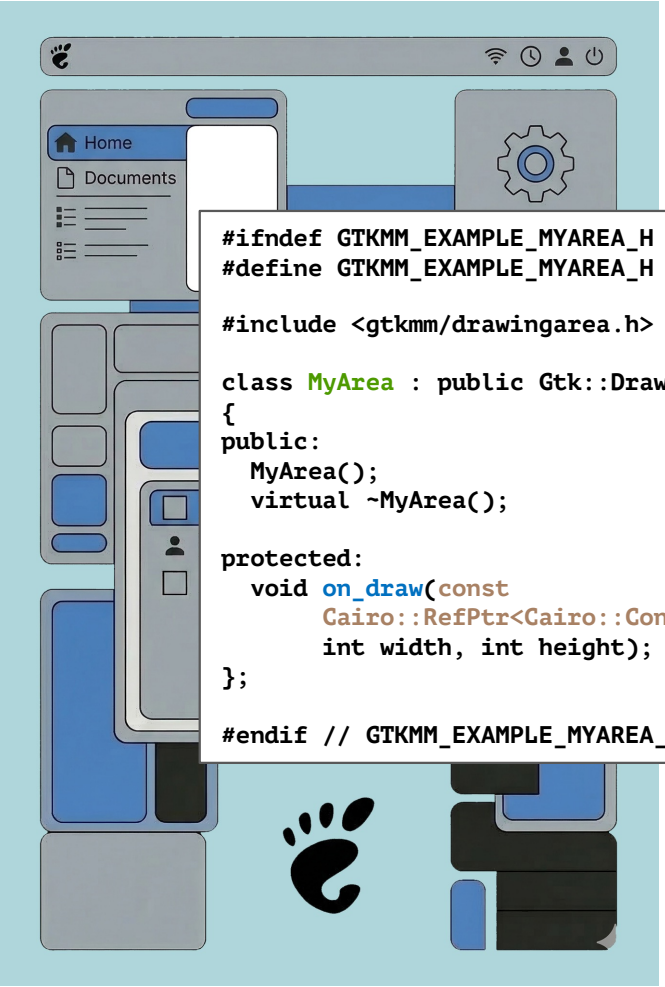
GTKmm4 : Exemple DrawingArea




```
#include "myarea.h"
#include <gtkmm/application.h>
#include <gtkmm/window.h>

class ExampleWindow : public Gtk::Window {
public:
  ExampleWindow();
protected:
  MyArea m_area;
};

ExampleWindow::ExampleWindow() {
  set_title("DrawingArea");
  set_child(m_area);
}

int main(int argc, char** argv) {
   auto app = Gtk::Application::create();
   return app-> make_window_and_run<ExampleWindow>(argc, argv);
}
```



GTKmm4 : Exemple DrawingArea



myarea.h

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();

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

#endif // GTKMM_EXAMPLE_MYAREA_H
```

myarea.cc

```
#include "myarea.h"
#include <cairomm/context.h>

MyArea::MyArea()
{
  set_draw_func(sigc::mem_fun(*this, &MyArea::on_draw));
}

MyArea::~MyArea(){}

void MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr
                     int width, int height)
{
  // changing the default background color to black
  cr->set_source_rgb(0.0, 0.0, 0.0); //mémorisé dans cr
  cr->paint();

  // center of the GTKmm window
  int xc(width/2), yc(height/2);

  cr->set_line_width(10.0); // idem mémorisé dans cr

  // draw red lines out from the center of the window
  cr->set_source_rgb(0.8, 0.0, 0.0); // idem mémorisation cr
  cr->move_to(0, 0);
  cr->line_to(xc, yc);
  cr->line_to(0, height);
  cr->move_to(xc, yc);
  cr->line_to(width, yc);
  cr->stroke();
}
```

rafael.pires@epfl.ch

Merci