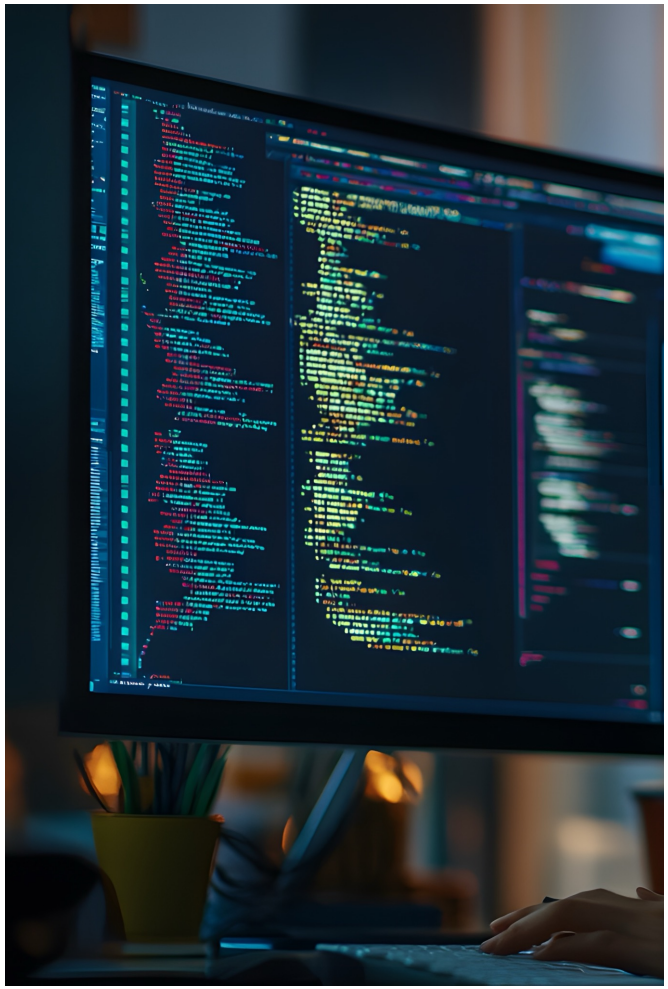
Programmation Orientée Projet
COM-112(a)

Semaine 4

Rafael Pires
rafael.pires@epfl.ch

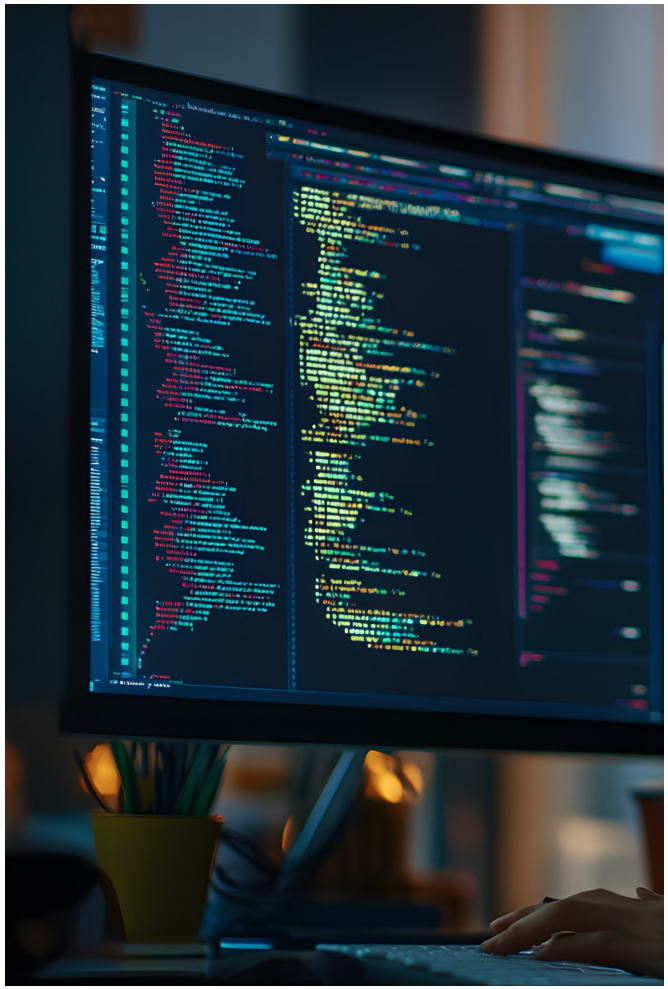
Lausanne, 13.03.2026

PoP 04



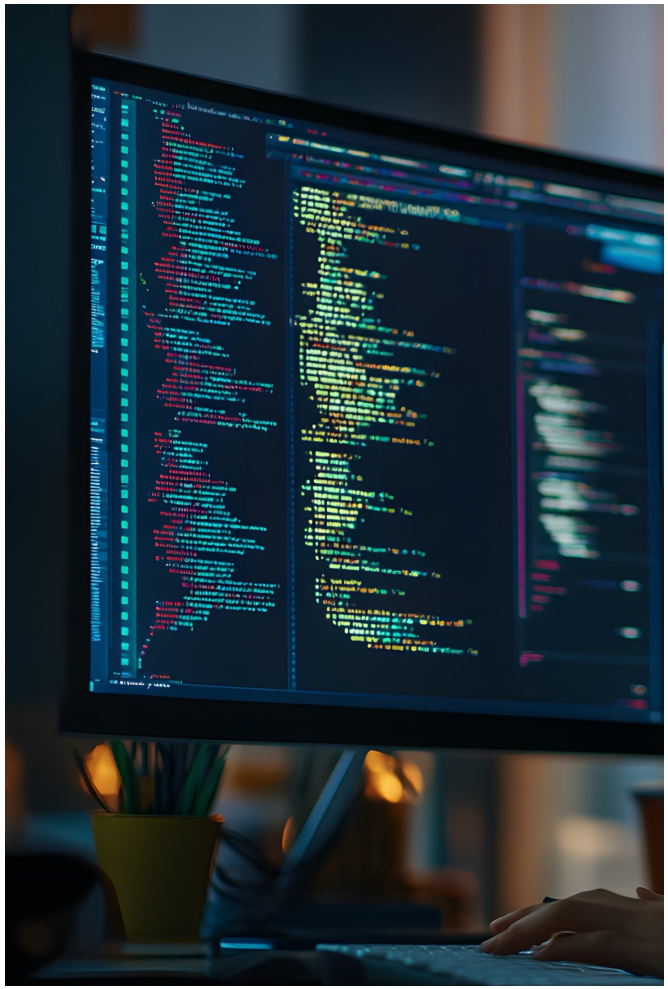
- MOOC :
  - Usage de `static`
  - Extension de la notion de surcharge
- Projet :
  - Paramètres en ligne de commande
  - Tableau dynamique
  - Type paramétré (série)

`static` : les 3 usages



- Durée de vie permanente
- N'est initialisée qu'une seule fois
  - Si la valeur n'est pas précisée, est initialisé avec des 0
  - Ou par le constructeur par défaut

`static` associé à une variable locale (1)



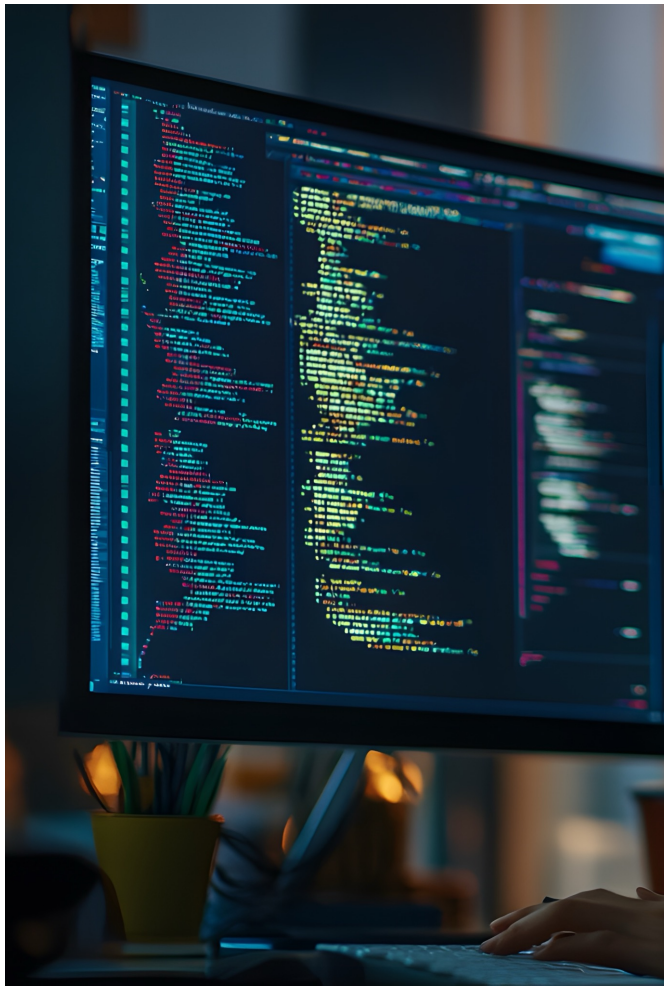
- Portée limitée au bloc de sa déclaration
- Conserve sa valeur d'un appel au suivant

```
unsigned int nb_appel() {
  static unsigned count(0);
  return ++count;
}


int main()
{
  cout << nb_appel() << endl;
  cout << nb_appel() << endl;
  cout << nb_appel() << endl;
…
```

Affichage

```
1
2
3
```

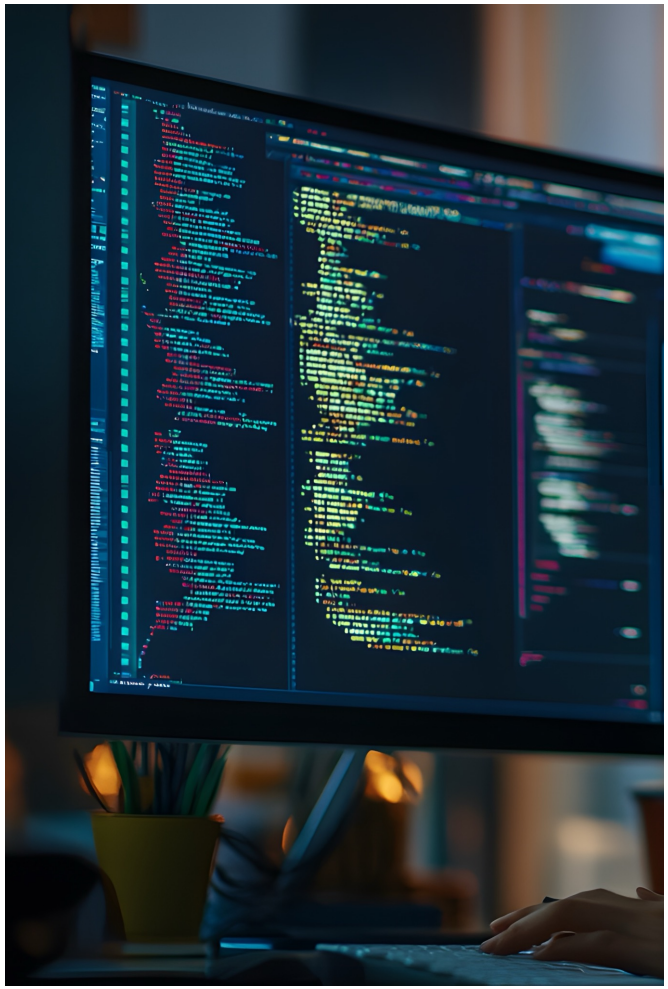
`static` associé à une variable globale (2)



myclass.cc

```
...
#include "myclass.h"

static vector<MyClass> tab; // vide

// externalisation de la définition
// des méthodes de la classe MyClass

MyClass::Myclass() ...
{...}

...
```

`static` associé à une variable globale (2)



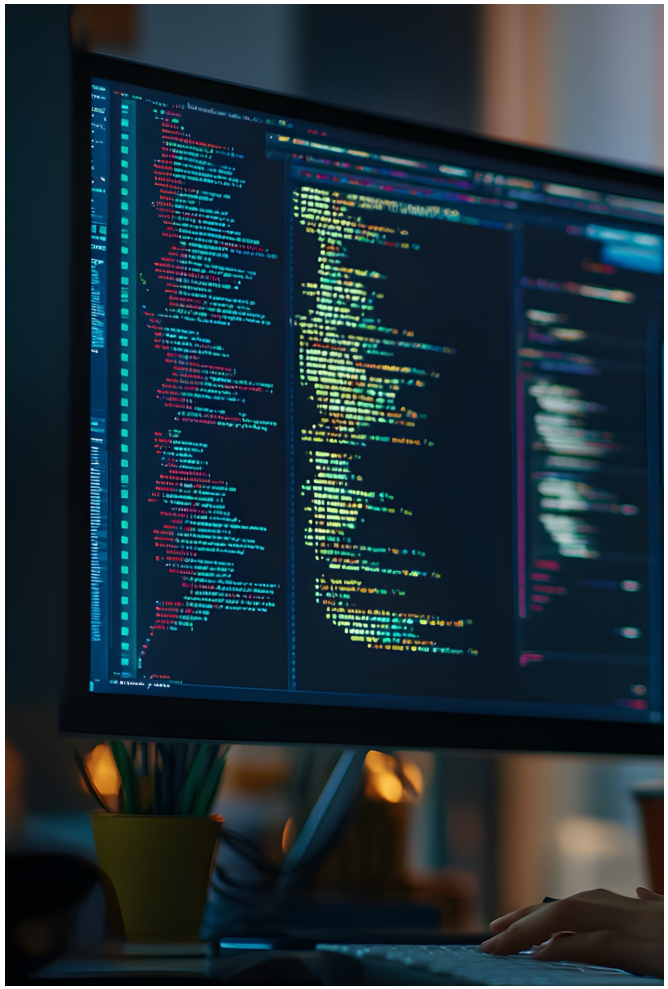
- Rend cette variable confidentielle au module
- Portée limitée au module

  (accessible par toutes les fonctions du module)
- Aussi utilisable pour une fonction

Usage :

- Ne jamais mettre dans l'interface d'un module
- Utile pour des constantes locales au module
- Sinon, à limiter au strict nécessaire

`static` associé à un attribut (3)



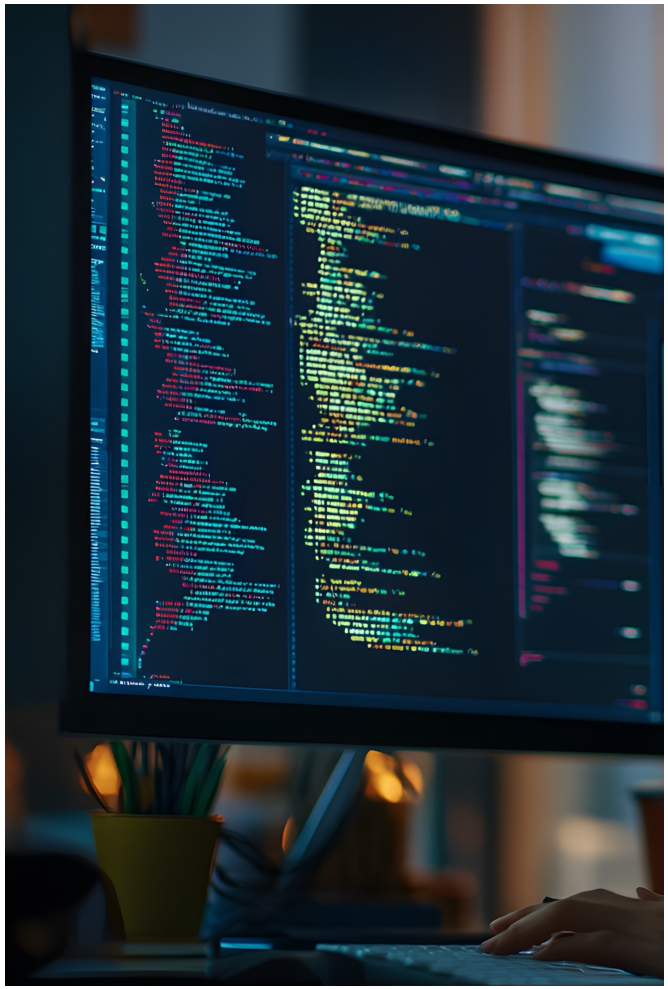
myclass.h

```
...
class MyClass {
private:
  static vector<MyClass> tab;
...
```

myclass.cc

```
...
#include "myclass.h"

vector<MyClass> MyClass::tab; // vide

// externalisation de la définition
// des méthodes de la classe MyClass

MyClass::Myclass() ...
{...}
...
```

`static` associé à un attribut (3)



- Portée limitée à la classe
  (accessible par toutes les méthodes de la classe)
- Doit être initialisé en dehors de la classe
- Aussi utilisable pour une méthode

Usage :

- Ne pas rendre `public`
- Utile pour des paramètres communs
- Sinon, à limiter au strict nécessaire

Quiz

Soit le module **rect** qui est testé dans la fonction main du fichier **prog.cc** :

prog.cc

```
// testing class Rect
#include <iostream>
#include "rect.h"

using namespace std;

int main()
{
    Rect r(1.2, 3.4);
    cout << r.surf() << endl;
    return 0;
}
```

rect.h

```
#ifndef RECT_H
#define RECT_H
// class Rect with one static attribute
class Rect
{
public:
    Rect(double width,double height);
    double surf() const;
private:
    static unsigned nb_rect;
    double width ;
    double height;
};

unsigned Rect::nb_rect(0);
#endif
```

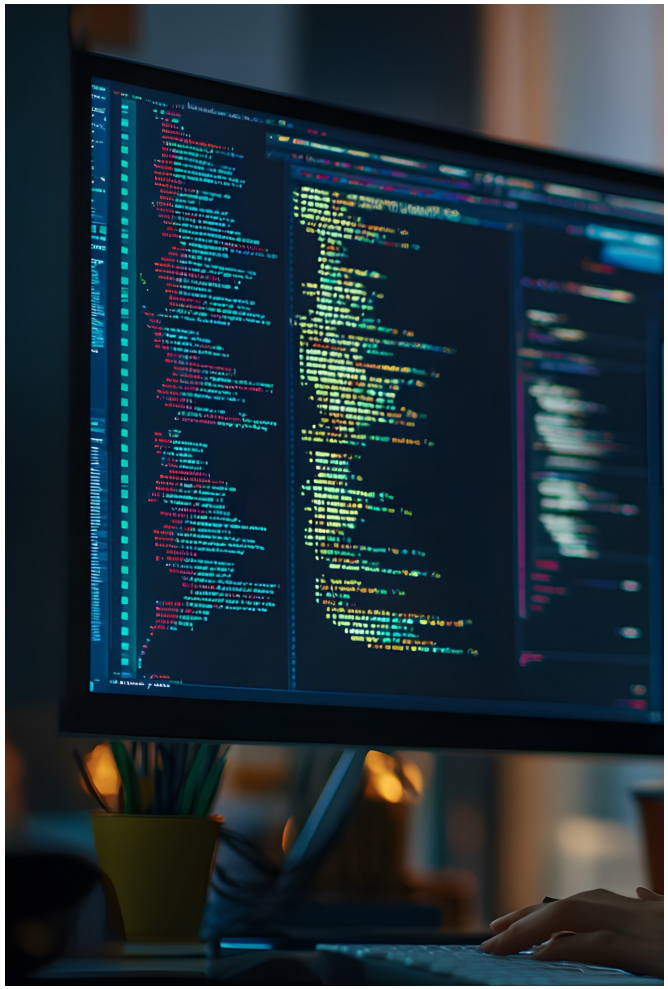
rect.cc

```
// class Rect with one static attribute
#include <iostream>
#include "rect.h"

using namespace std;

Rect::Rect(double width, double height)
    :width(width),height(height) {++nb_rect;}

double Rect::surf() const
{
    return width*height;
}
```

Donner la réponse correcte si on lance : `g++ -std=c++11 prog.cc rect.cc -o prog`

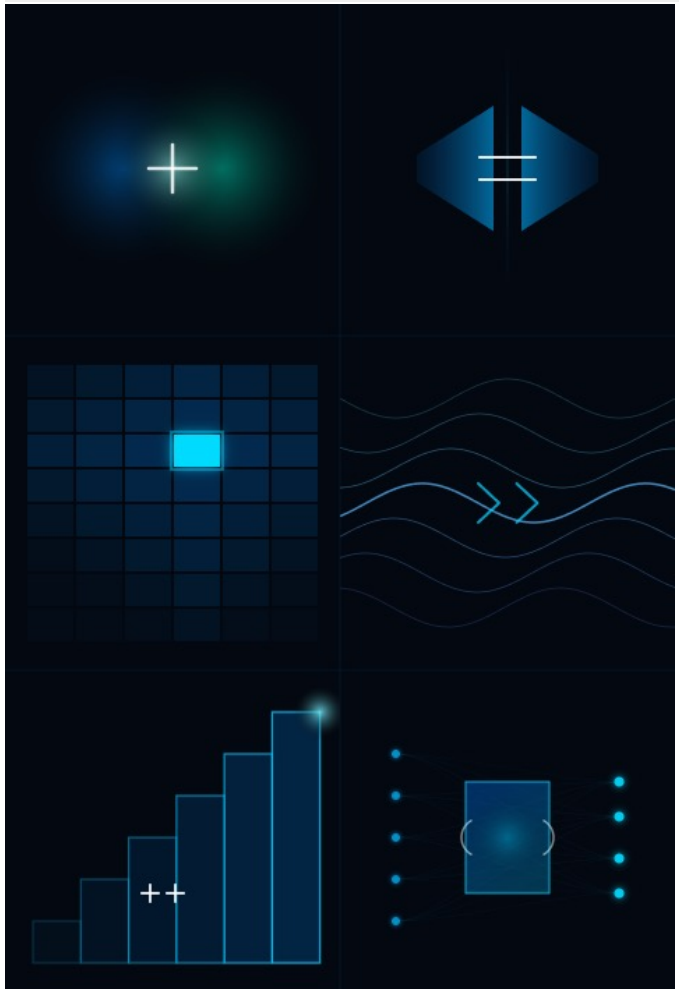
A. Erreur de **compilation** parce que l'attribut static est défini deux fois
B. Erreur de **l'édition de liens** parce que l'attribut static est défini deux fois
C. L'attribut static est défini deux fois ; cependant un exécutable est produit et fonctionne bien
D. L'attribut static est défini une seule fois ; l'exécutable est produit et fonctionne bien

PoP 04



- **MOOC :**
  - Usage de `static`
  - Extension de la notion de surcharge
- **Projet :**
  - Paramètres en ligne de commande
  - Tableau dynamique
  - Type paramétré (série)

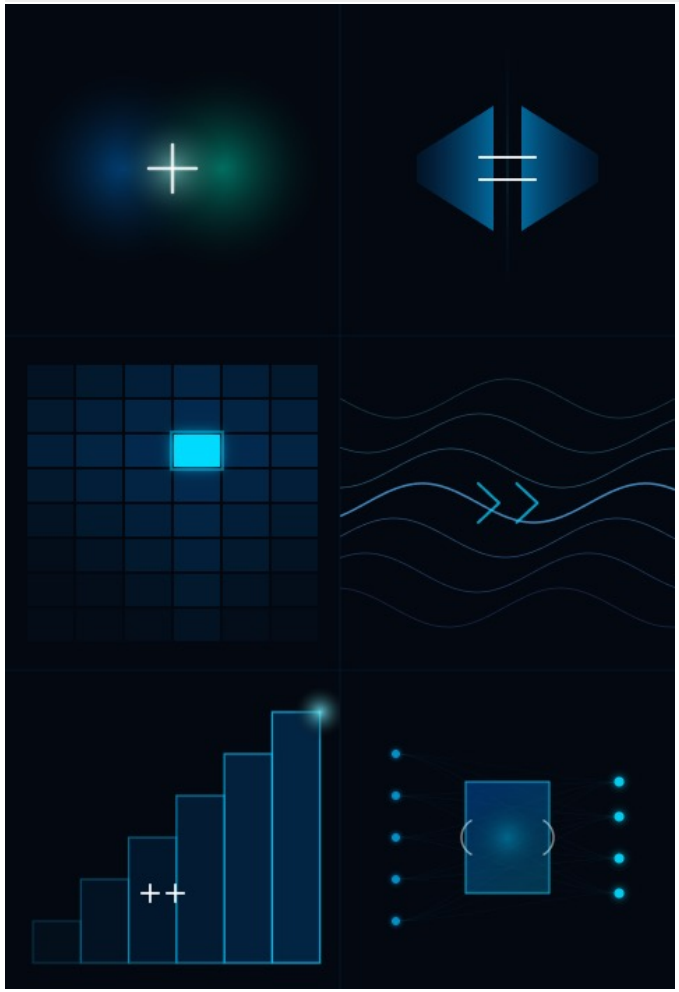
MOOC semaine 3 : extension de la notion de surcharge



- Déjà connu pour les fonctions et les méthodes
- Règle de base
  - Si plusieurs fonctions/méthodes ont le même nom, le compilateur sait laquelle doit être appelée car il peut les différencier grâce à leur signature
  - La résolution de surcharge est basée sur le nombre et les types des paramètres. Le type de retour n'en fait pas partie.

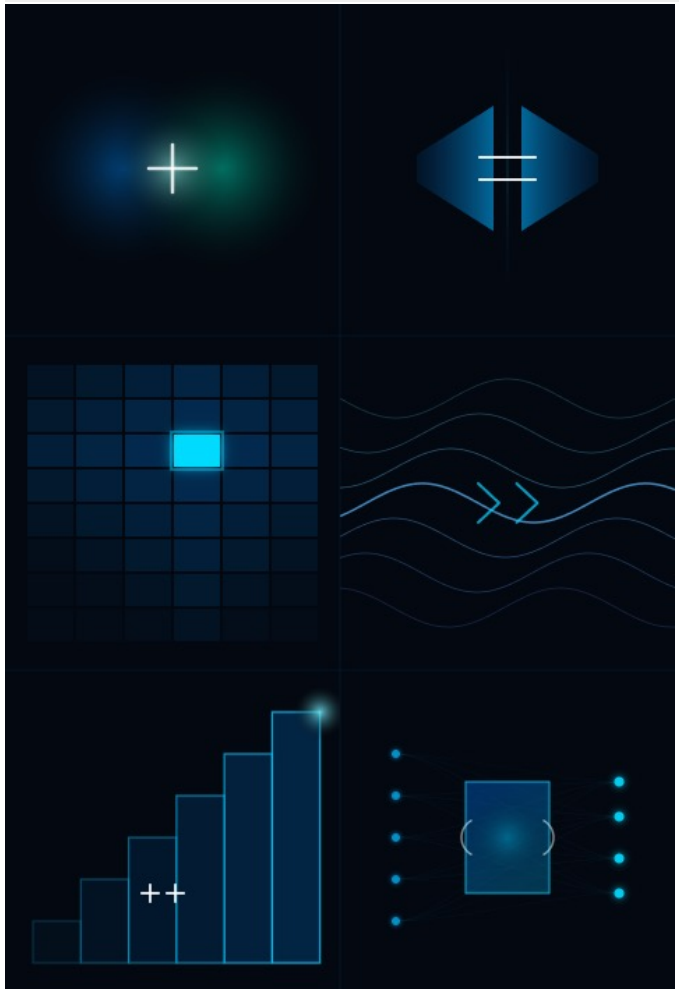
```
int affiche(int);    // ok
double affiche(int); // erreur car même signature
int affiche(double); // ok
```

Surcharge des opérateurs



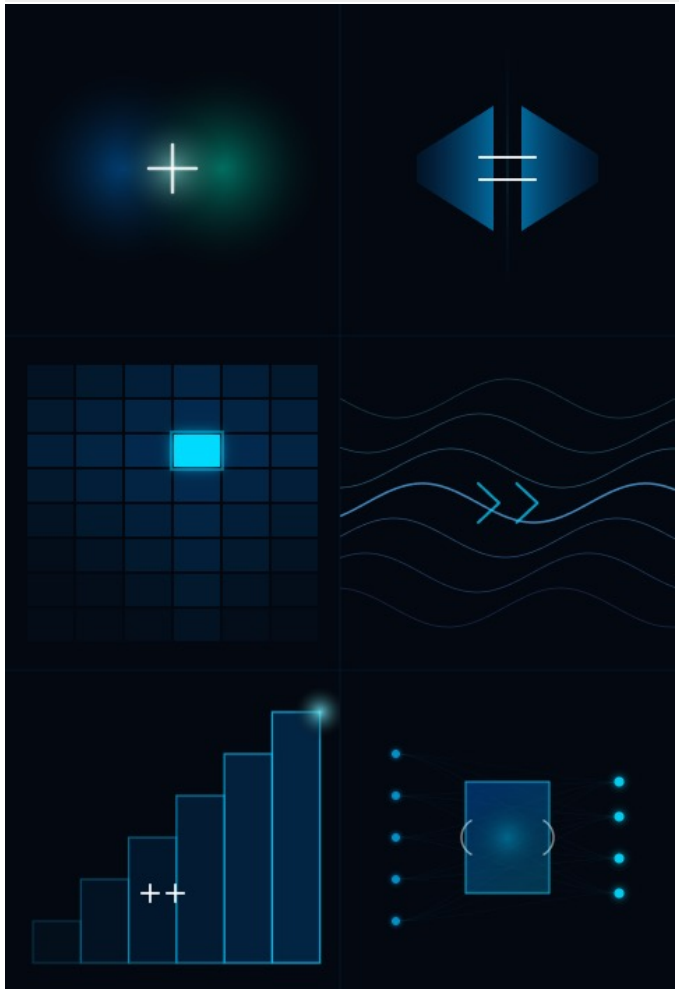
- Motivation
  - Implémenter une solution avec une plus grande concision d'écriture
  - Bénéficier de la sémantique familière associée aux symboles des opérateurs pour l'élargir au-delà des types de base
- Inconvénient
  - Risques de confusion si mal employé (obfuscation du code)
- Conseil
  - Mettre en œuvre la réutilisation pour la surcharge des opérateurs ayant un lien sémantique, par exemple `==` et `!=` ou `+` et `+=`

Surcharge interne ou externe ?



	méthode de classe	fonction
a+b	a.operator+(b)	operator+(a, b)
a += b	a.operator+=(b)	operator+=(a, b)
a = b	a.operator=(b)	
a == b	a.operator==(b)	operator==(a,b)
cout << b		operator<<(cout, b)
++a	a.operator++()	operator++(a)

Surcharge interne ou externe ?



méthode de classe

fonction

Préférable

- Si l'opérande gauche est modifié
- Si l'accès aux attributs est requis

ex. : modifie **a** donc surcharge interne

```
a += b     a.operator+=(b)
```

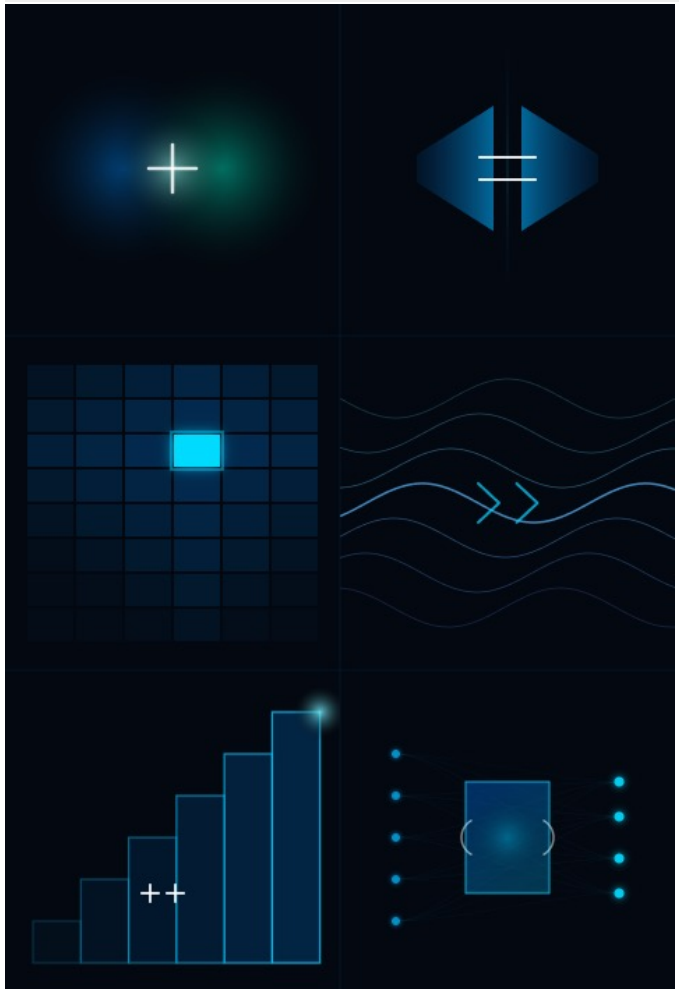
Obligatoire si l'opérande gauche :

- Est un type de base
- N'appartient pas à la classe dont on veut surcharger l'opérateur

ex. : **cout** et **ostream**

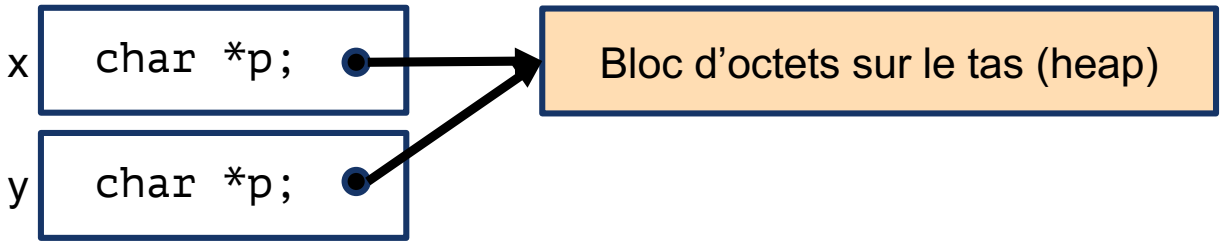
```
cout << b    operator<<(cout, b)
```

Surcharge interne de l'opérateur d'égalité



- **Copie superficielle**

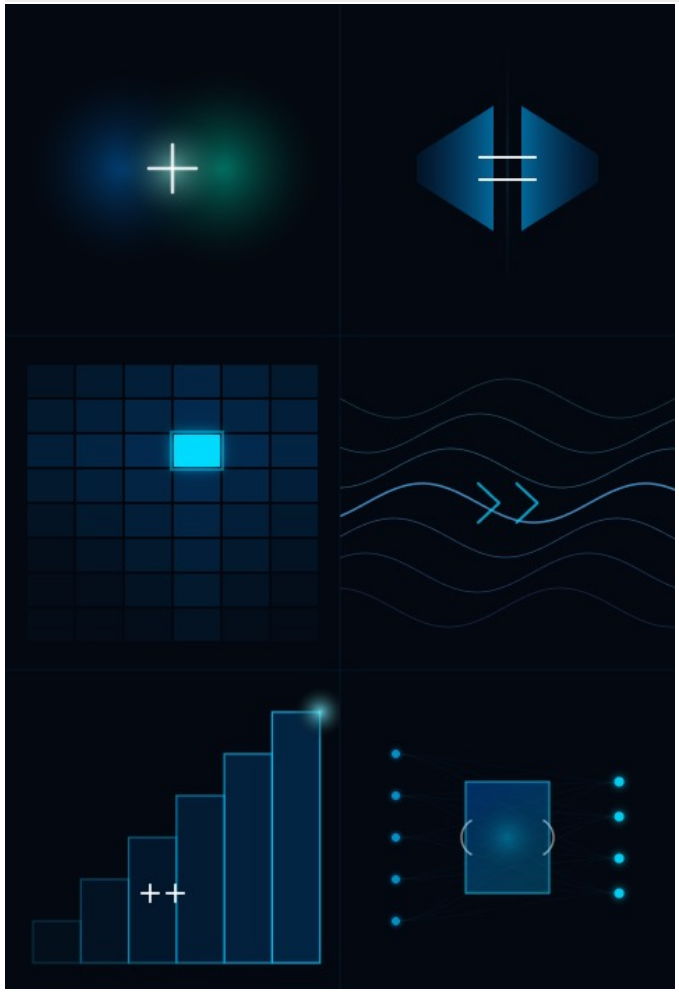
```
CADS <==> Classe avec allocation dynamique et copie superficielle
CADS(50);  // constructeur avec allocation
CADS y(x); // copie superficielle
```




```
bool operator==(CADS const& b)
{
  return p == b.p;
}
```

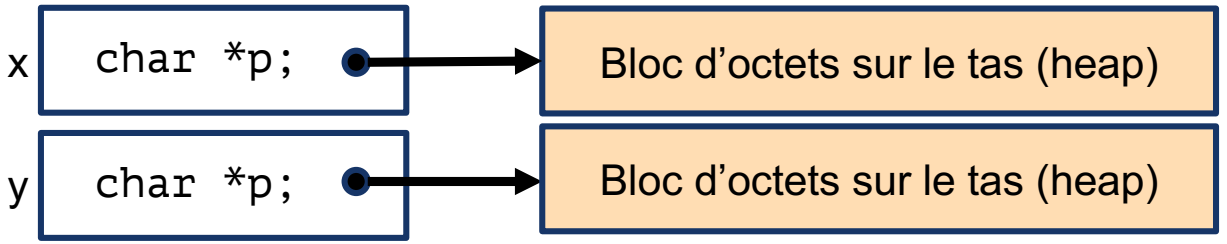
on obtient **true** pour l'expression **x == y** avec le scénario de copie superficielle car l'attribut de x et de y ont la même valeur d'adresse de bloc (OK).

Surcharge interne de l'opérateur d'égalité



- **Copie profonde**

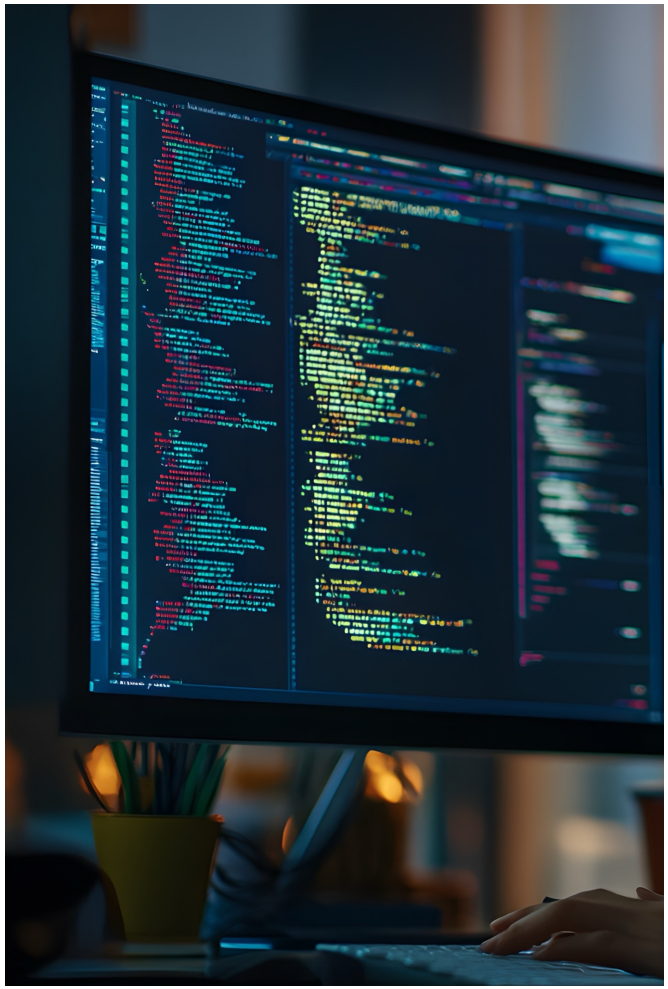
```
CADP <==> Classe avec allocation dynamique et copie profonde
CADP(50);  // constructeur avec allocation
CADP y(x); // copie profonde
```




```
bool operator==(CADP const& b)
{
  return p == b.p;
}
```

on obtient **false** pour l'expression **x == y** avec le scénario de copie profonde car l'attribut de x et de y n'ont pas la même valeur d'adresse de bloc

Rappel : Règle à mémoriser



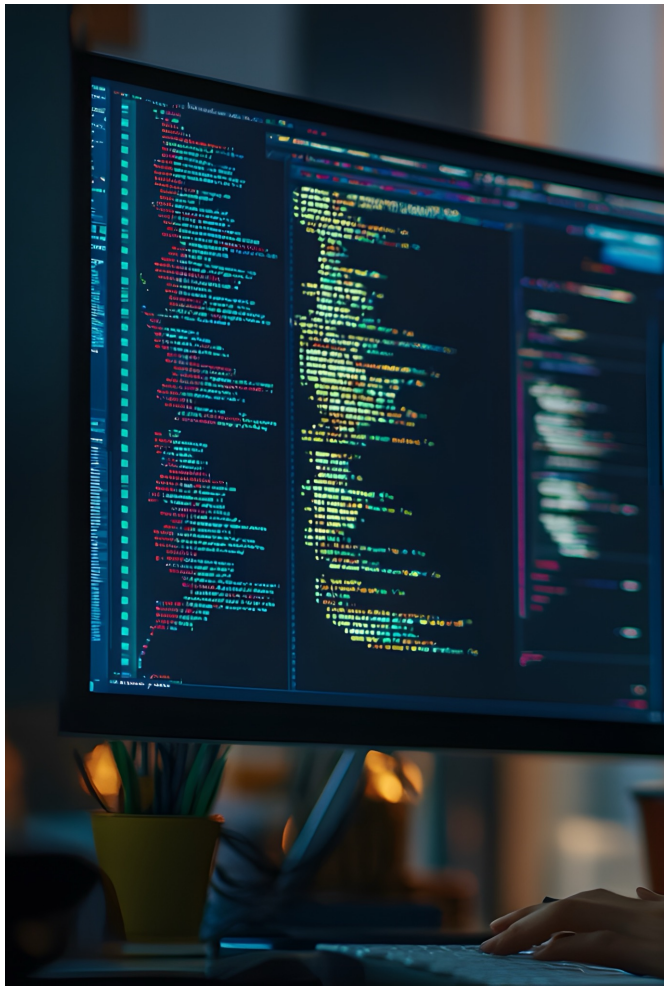
- Si on modifie l'une des trois méthodes ci-dessous, **il faut** vérifier si les deux autres doivent être aussi adaptées

  - Constructeur de copie
  - Destructeur
  - Opérateur d'affectation

⚠ Allocation dynamique : Copie/comparaison superficielle/profonde

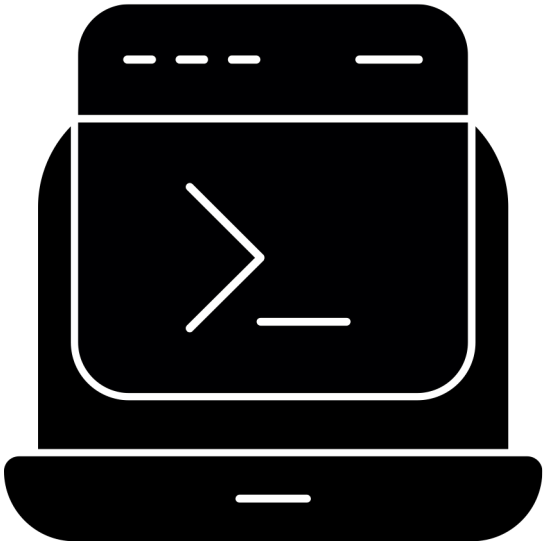
⚠ Ressources OS (système d'exploitation) : fichiers, réseau

PoP 04



- **MOOC :**
  - Usage de `static`
  - Extension de la notion de surcharge
- **Projet :**
  - **Paramètres en ligne de commande**
  - **Tableau dynamique**
  - **Type paramétré (série)**

Paramètres en ligne de commande

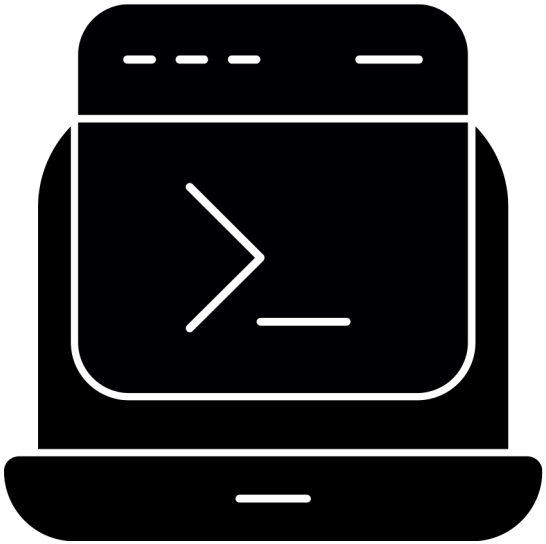


main.cpp

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Nombre de paramètres : " << argc << std::endl;

  for (int i = 0; i < argc; i++) {
    std::cout << "argv[" << i << "] = " << argv[i] << std::endl;
  }

  return 0;
}
```

```
g++ -o programme main.cpp
./programme bonjour 42 monde
Nombre de paramètres : 4
argv[0] = ./programme
argv[1] = bonjour
argv[2] = 42
argv[3] = monde
```

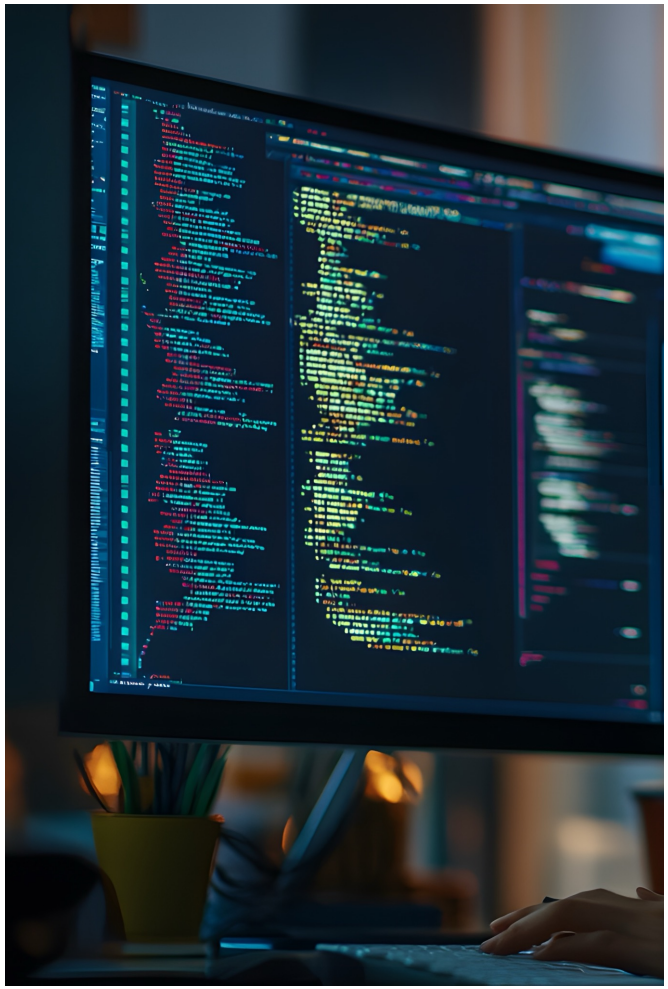
Paramètres en ligne de commande



- `argc` (*argument count*) — le nombre de paramètres, **toujours ≥ 1**
- `argv` (*argument vector*) — le tableau de chaînes de caractères
- `argv[0]` est **toujours** le nom du programme lui-même
- Les paramètres sont des `char*` (chaînes C) — pour utiliser un nombre, il faut le convertir avec `std::stoi()` ou `std::stod()`

```
int n = std::stoi(argv[1]); // convertit "42" en entier 42
```

PoP 04



- **MOOC :**
  - Usage de `static`
  - Extension de la notion de surcharge
- **Projet :**
  - Paramètres en ligne de commande
  - **Tableau dynamique**
  - Type paramétré (série)

Tableau dynamique : vector

```
#include <vector>       // Fait partie des outils de type container
vector<double> tab(5); // Initialisation avec le motif binaire nul
tab.size()             // Un vector connaît sa taille
tab[i]                 // Accès en O(1) à tout élément du vector
```

- Un **vector** est particulièrement efficace pour ajouter/enlever le dernier élément (coté back) avec **push_back** et **pop_back**.
- Cette fonctionnalité permet facilement de mettre en œuvre le concept de pile (LIFO = Last In, First Out)

```
tab.push_back(x); // ajoute la valeur x ici
tab.pop_back(); // enlève cette dernière valeur
```

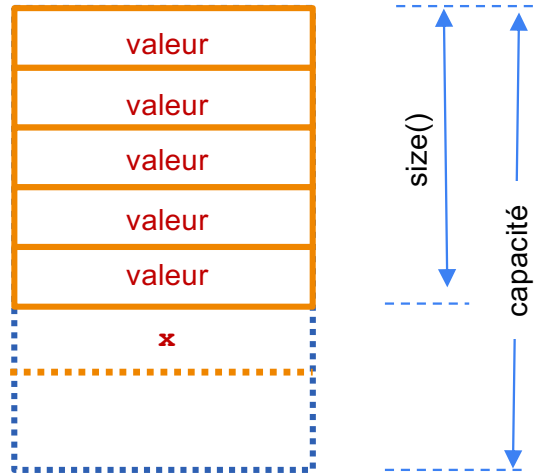


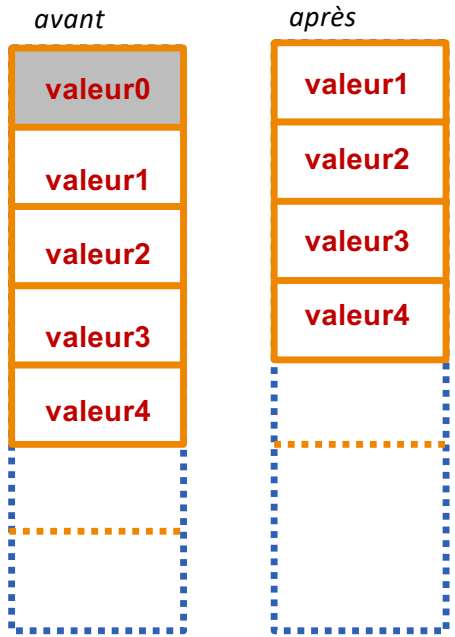
Tableau dynamique : `vector`

- **Point faible 1** : une méthode `insert` permet d'insérer un ou plusieurs éléments à l'intérieur du vecteur MAIS cela implique un coût de décalage de tous les éléments qui suivent. Même problème avec la méthode `erase` pour enlever un ou plusieurs éléments.

- **Point faible 2** : il n'existe PAS de méthode pour ajouter/enlever le premier élément (coté front). Utiliser le container `deque` (double-ended queue). Cet outil permet de mettre en œuvre le concept de file d'attente (queue).

- **Observation 1** : les méthodes `insert` et `erase` ne doivent pas modifier l'ordre des éléments déjà présent dans un `vector`. C'est pourquoi elles induisent un coût de décalage des éléments qui suivent dans le `vector`.

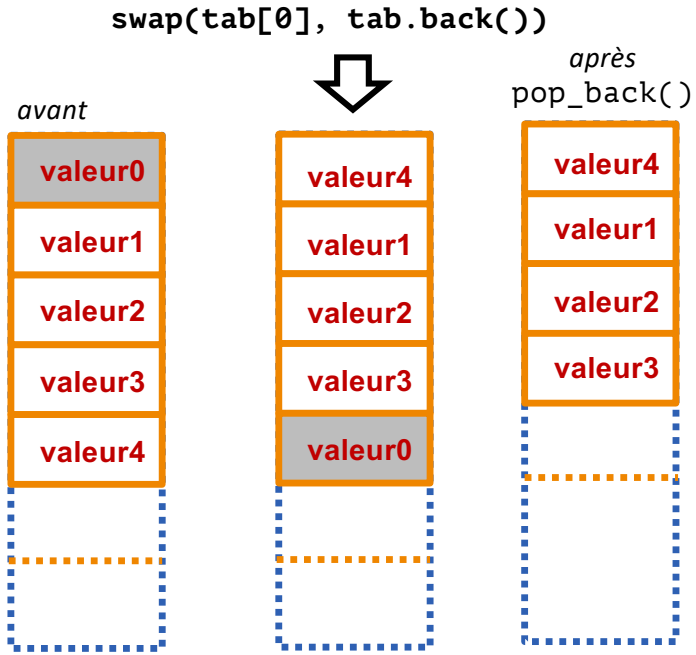
- Conséquence négative: coût linéaire O(`tab.size()`)



Ex: utilisation de `erase` pour supprimer le premier élément de **valeur0**

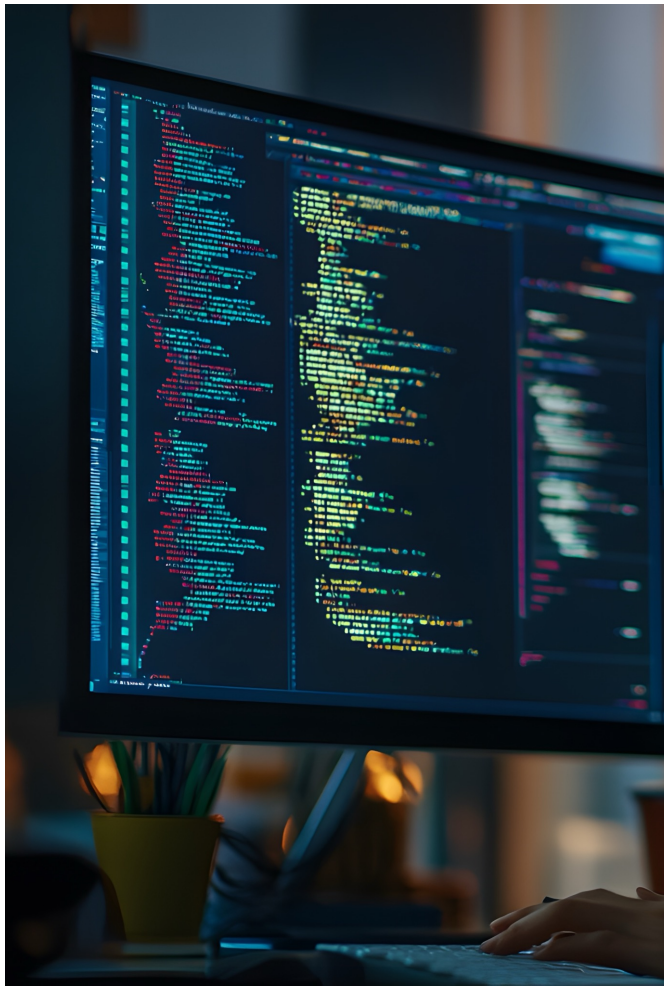
Tableau dynamique : `vector`

- **Observation 2 :** si le tableau dynamique n'a pas besoin d'être trié, alors on peut réduire drastiquement l'ordre de complexité des opérations d'ajout et de suppression d'un élément.
- Pour la suppression, il suffit d'utiliser la méthode `swap` pour échanger la valeur de l'élément à supprimer avec celle du dernier élément, puis d'appeler `pop_back()`.
- Conséquence positive: coût constant **O(1)**
- Pour l'ajout, il suffit d'appeler `push_back(valeur)`.
  Conséquence positive: généralement à coût constant **O(1)**



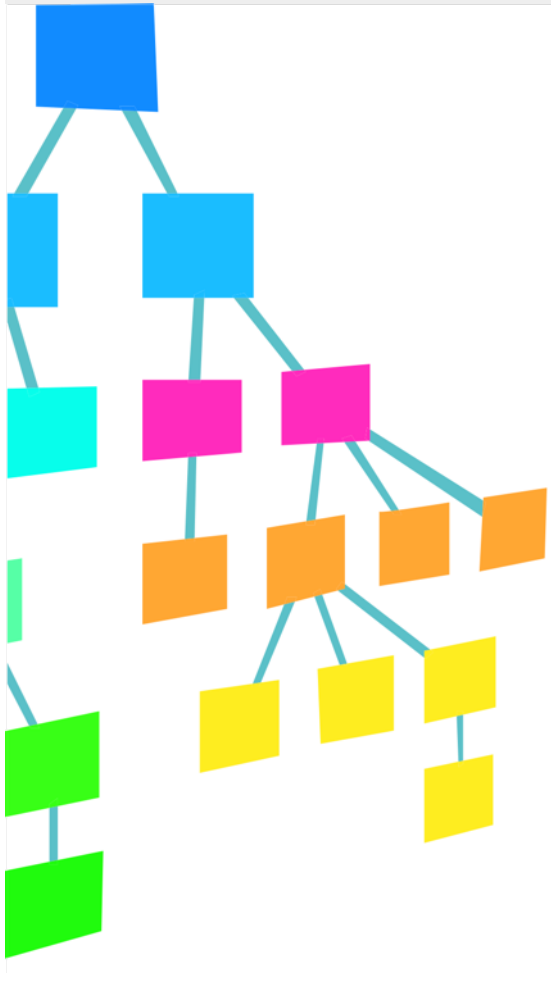
Ex: suppression du premier élément du vector **tab** de valeur **valeur0**

PoP 04



- **MOOC :**
  - Usage de `static`
  - Extension de la notion de surcharge
- **Projet :**
  - Paramètres en ligne de commande
  - Tableau dynamique
  - **Type paramétré (série)**

Type paramétré



- La classe contient un attribut « catégorie » qui est ensuite utilisé au niveau des méthodes pour sélectionner le code exécuté pour chaque catégorie.

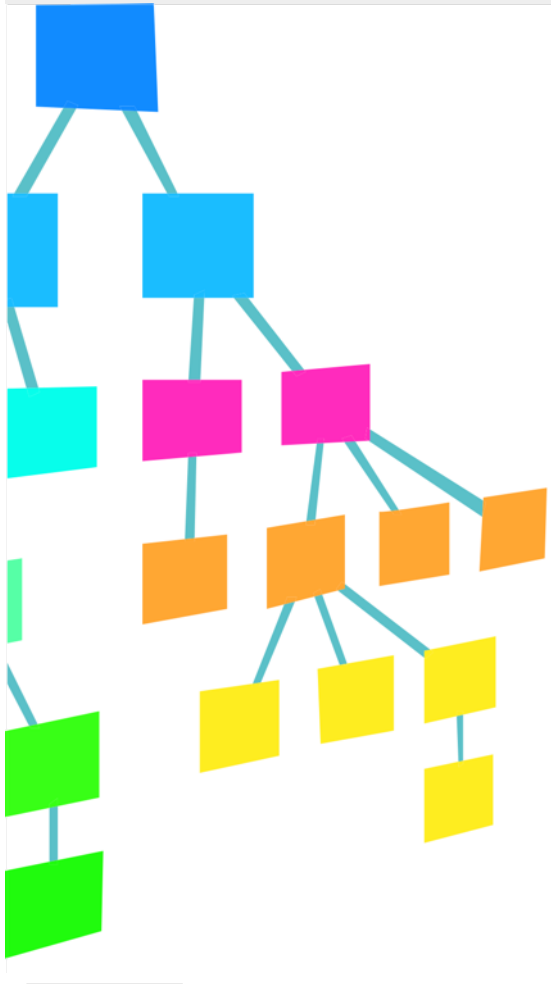
form.h

```
enum Category {CIRCLE, SQUARE,RECTANGLE};
…
class Form
{
public:
  Form(Category c=CIRCLE, double x=0, double y=0, double p1=0);
  Form(Category c, double x, double y, double largeur, double hauteur );
  …
  void affiche();
  …
private:
  Category category;
  double x, y; // centre de la Form
  std::vector<double> param{std::vector<double>(1,0.)}; //un param par défaut
};
```

Constructeur pour CIRCLE ou SQUARE:
p1 est le rayon pour CIRCLE et le coté pour SQUARE
Ce constructeur sert aussi de constructeur par défaut

Constructeur pour RECTANGLE

Type paramétré

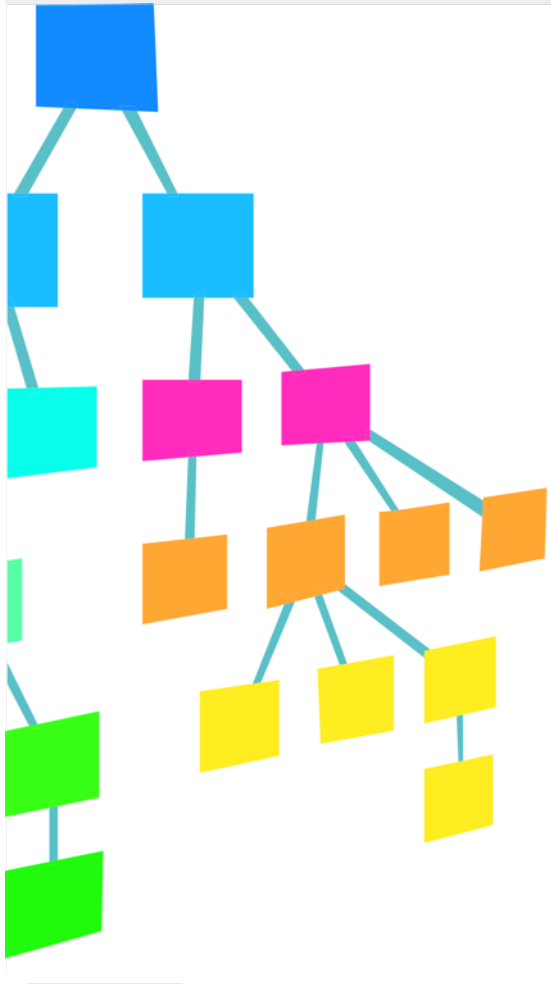


- Ce constructeur de `CIRCLE` ou de `SQUARE` vérifie l'adéquation de la catégorie et de la valeur transmise pour le paramètre (rayon / coté)

form.cc

```
#include "form.h"
…
Form::Form(Category c, double x, double y, double p1)
  : category(c), x(x), y(y) {

  switch(category) {
    case CIRCLE:
    case SQUARE:
      param[0] = p1;
      if(param[0] < 0.) {
        cout << "Form: incorrect parameter value" << endl;
        exit(EXIT_FAILURE);
      }
      break;

    default:
      cout << "Form: incorrect category" << endl;
      exit(EXIT_FAILURE);
  }
}
```

Type paramétré

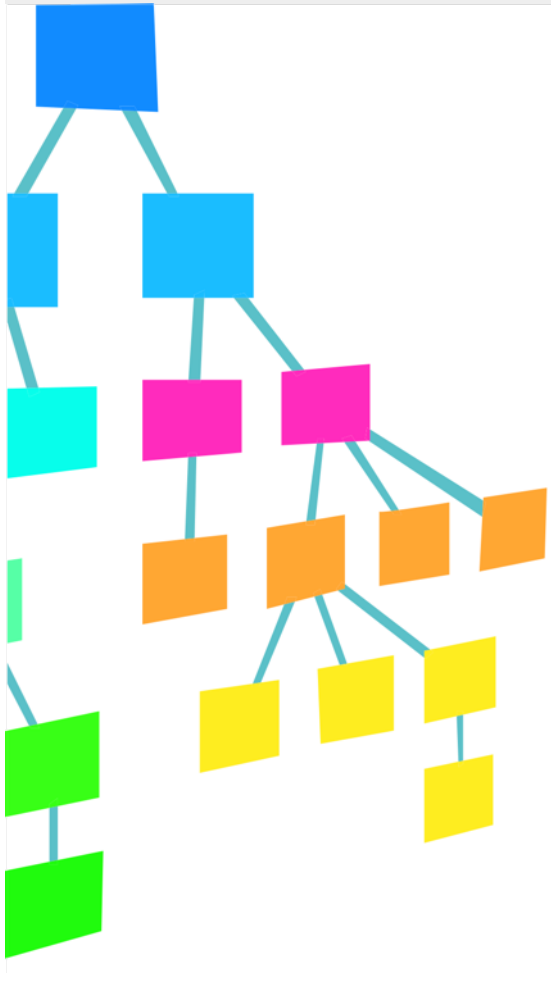


form.cc

```
#include ”form.h"
…
void Form::affiche() {
  switch(category) {
  case CIRCLE:
    cout << "CIRCLE with center:(" << x << ", " << y
         << " ) and radius " << param[0] << endl;
    break;

  case SQUARE:
    cout << "SQUARE with center:(" << x << ", " << y
         << " ) and side " << param[0] << endl;
    break;

  case RECTANGLE:
    cout << "RECTANGLE with center:(" << x << ", " << y << " ), width: "
         << param[0] << " and height: " << param[1] << endl;
    break;

  default:
    cout << "Form: incorrect category" << endl;
    exit(EXIT_FAILURE);
  }
}
```

Une méthode typique devra toujours contenir un switch sur l’attribut `category`

Type paramétré

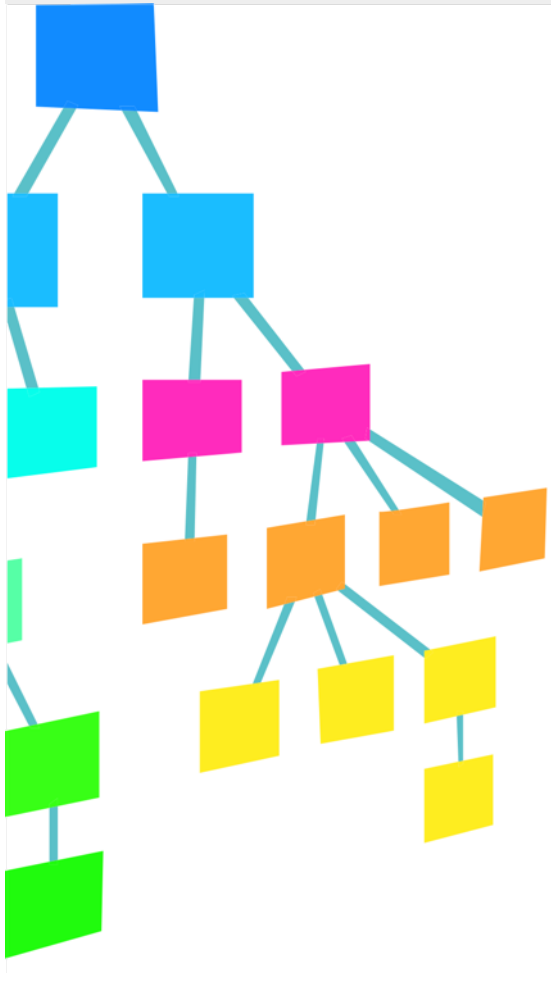


- Des instances de différentes catégories peuvent être stockées dans un même conteneur

main.cc

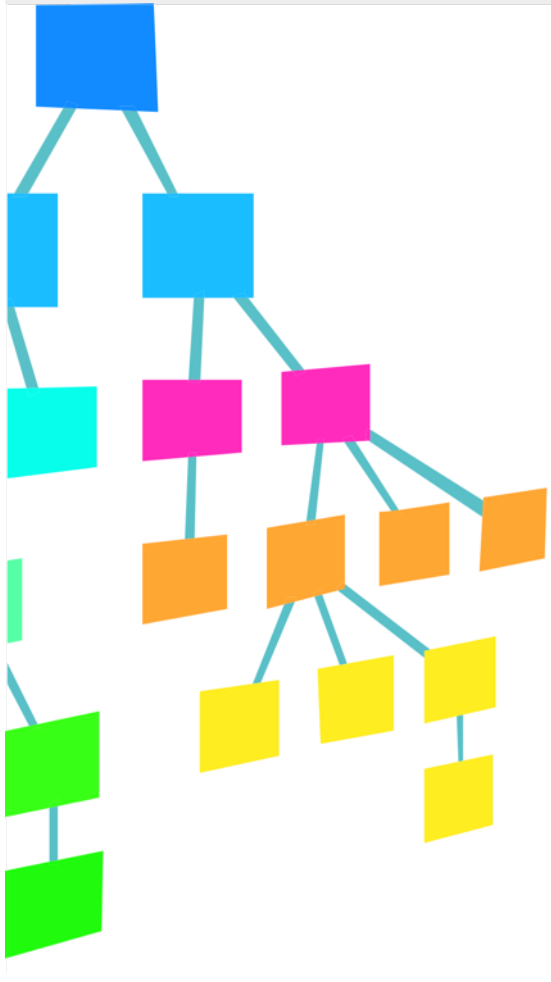
```
#include "form.h"
…
int main() {
  vector<Form> tab;
  …
  // ici du code qui ajoute des instances Form
  // de différentes catégories dans le vector tab
  …

  // unique boucle qui délègue à la classe Form
  // le détail des actions pour chaque méthode

  for(const auto& s : tab)
    s.affiche();
  …
}
```

Type paramétré : faiblesses



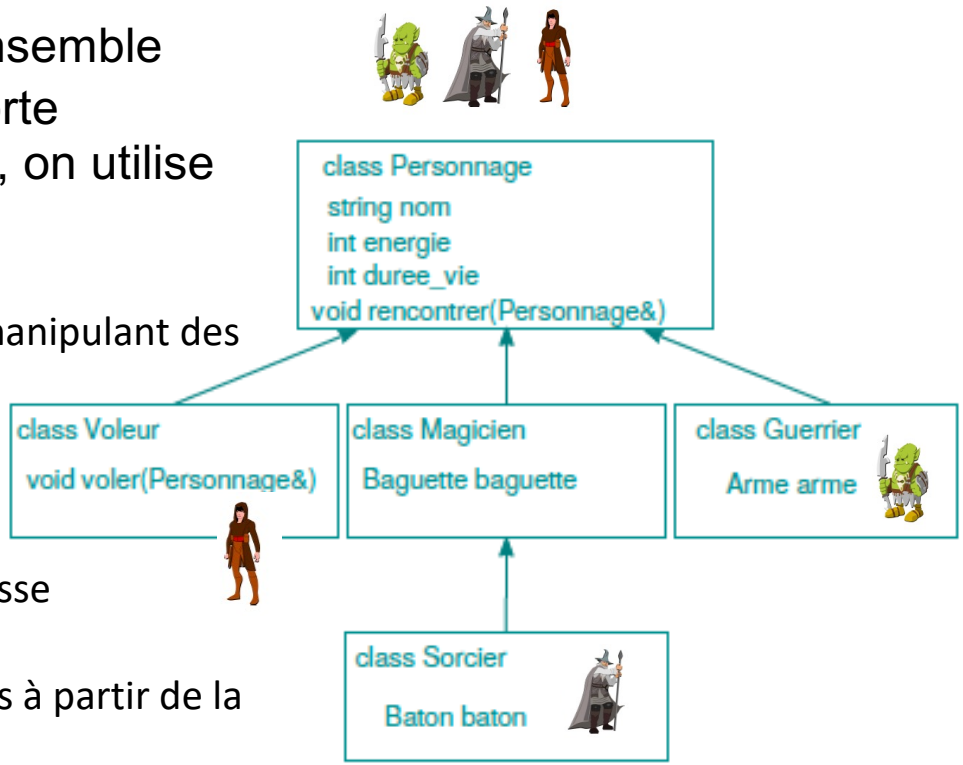
- Pourquoi une seule classe pour toutes les catégories pose problème ?
  - Validation globale uniquement : un bug dans une catégorie bloque toutes les autres
  - Maintenance risquée : modifier une méthode peut casser du code déjà validé
  - Extension difficile : ajouter un type (ex. `TRIANGLE`) impose de modifier chaque `switch` sur `category`
- Une classe par catégorie ?
  - Mieux, mais : code dupliqué entre classes similaires, et impossible de mélanger les types dans un même conteneur.
- La solution : une hiérarchie de classes *(MOOC sem. 4)*

Héritage : teaser



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

  - ❖ Eviter de dupliquer du code manipulant des propriétés communes
  - ❖ Partager un sous-ensemble de propriétés/méthodes communes dans une superclasse
  - ❖ Définir des classes spécialisées à partir de la superclasse



Résumé Cours 4

- `static` associé à
  - une variable locale : portée limitée au bloc, conserve sa valeur d'un appel au suivant
  - une variable globale : rend la variable confidentielle au module, portée limitée au fichier `.cc`
  - un attribut de classe : partagé entre toutes les instances, doit être initialisé en dehors de la classe
- Surcharge des opérateurs : interne (méthode de classe, préférable si l'opérande gauche est modifié ou si l'accès aux attributs est requis) vs externe (fonction libre, obligatoire si l'opérande gauche est un type de base ou n'appartient pas à la classe)
- Opérateur d'égalité `==` : en copie superficielle, deux objets partageant le même bloc mémoire sont égaux ; en copie profonde, deux blocs distincts donnent false même si le contenu est identique
- `argc`/`argv` permettent de passer le fichier de test en argument sur la ligne de commande
- `vector` est efficace en $O(1)$ pour `push_back`/`pop_back`, mais coûteux en $O(n)$ pour `insert`/`erase` ; l'astuce `swap` + `pop_back` évite ce coût quand l'ordre n'importe pas
- Le type paramétré (une classe, un attribut catégorie, des `switch`) gère des entités hétérogènes dans un même conteneur, mais ses limites de maintenance et d'extensibilité motivent l'héritage (semaine prochaine)

rafael.pires@epfl.ch

Merci