# MOOC semaine 5
# Polymorphisme d'inclusion

**Objectif:** Bénéficier du mécanisme de specialisation par derivation de classe en dépassant la fragmentation des types que cette approche introduit

## Plan:

- Ce que le polymorphisme permet d'éviter…
- Hiérarchie de classe et fragmentation des types
- La resolution **statique** des liens
- Pointeur et resolution **dynamique** des liens
- Intérêt: traitement d'une collection hétérogène

## Ce que le polymorphisme permet d'éviter:

Travailler avec une classe unique qui traite une grande variété d'éléments à l'aide d'un attribut de « type d'élément ». Ex: la classe **Form** vue en Série4

## Pourquoi est-ce une pratique à déconseiller ?

Le code est difficile à maintenir car de nombreuses méthodes utilisent un switch sur l'attribut « type d'élément ». Etendre le code avec un nouveau type d'élément présente un risque d'introduire des bugs sur du code déjà validé.

## N'est-ce pas déjà résolu avec une hiérarchie de classe?

En introduisant des classes dérivées on introduit des types distincts. Du coup on ne peut plus construire un ensemble d'éléments homogènes comme dans l'approche initiale.

# RAPPEL Notion de type paramétré (série4 PoP)

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

Exemple: cette classe **Form** contient un attribut **Category** défini avec **enum** et qui peut prendre l'une des valeurs (entière) des trois symboles : CIRCLE, SQUARE, RECTANGLE

```
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

# RAPPEL Notion de type paramétré : avantage principal

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

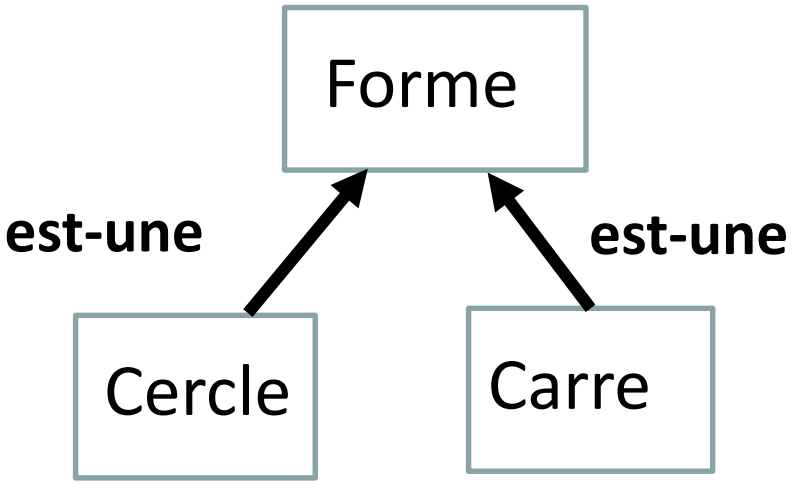
main.cc

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

# L'héritage est un bon début




… mais nous n'avons plus un unique type pour caractériser une instance de cette hiérarchie de classes:

`Forme, Cercle, Carre`…

Si aucune précision n'est apportée dans la définition de la hiérarchie de classes (<=>MOOC sem4), le compilateur effectue une résolution ***statique*** des liens :

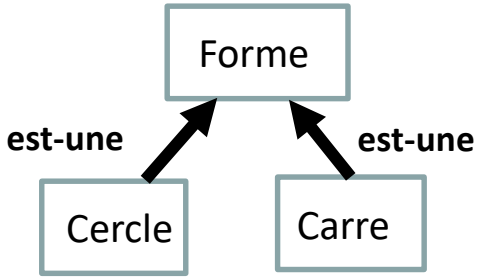
Le ***type*** des variables détermine la nature des méthodes qui sont appelées:

- Une variable de la superclasse appelle les méthodes de la superclasse
- Une variable d'une classe dérivée appelle les méthodes redéfinies dans la classe dérivée (**masquage/substitution**) ou, implicitement, celles de la superclasse s'il n'y a pas eu de redéfinition.



forme_res_statique.cc

# Héritage et Affectation




Une instance d'une classe dérivée peut être affectée à une classe parente:

```
Forme f;
Cercle c;
f = c ; // un cercle est une forme
```

Même chose pour un pointeur:

```
Forme* pf(nullptr);
Cercle* pc(&c);
pf = pc ; // un cercle est une forme
```

Cependant, avec la résolution **statique**, on perd le lien/les informations spécialisées des classes dérivées

Dans l'exemple **forme_res_statique.cc** le paramètre **Forme&** reçoit l'équivalent de l'adresse d'une **Forme** ou d'un **Cercle** selon les instructions,

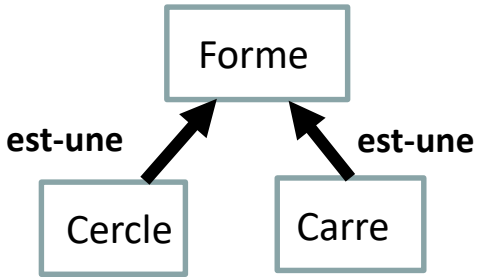
MAIS dans tous les cas, le compilateur ne voit qu'un type **Forme&** et il établit toujours le lien vers la méthode de la classe **Forme**.

# RAPPEL : occupation mémoire d'une instance et affectation

L'occupation mémoire d'une instance dépend **des types des attributs** (même règles d'alignement que pour les structures / revoir cours struct).

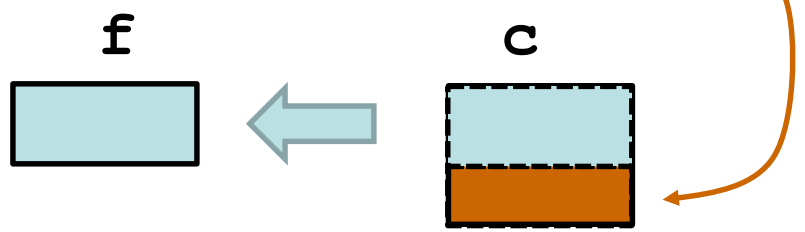
La spécialisation d'une classe dérivée se traduit souvent **par l'ajout d'attributs.**

Chaque classe dérivée ajoute une « tranche » mémoire pour ses attributs, qui vient à la suite de la zone mémoire des attributs de la classe parente.




```
Forme f;
Cercle c;
f = c ;
```

De ce fait l'affectation d'une instance d'une classe dérivée à une instance d'une classe parente se traduit par la **perte des attributs traduisant la spécialisation de la classe dérivée *(object slicing)***

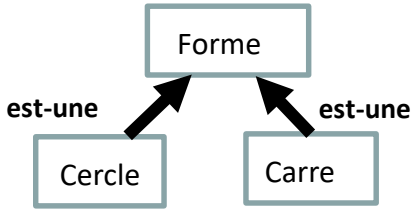



```
f = c ;
```

# Le pointeur = une clef pour la résolution dynamique

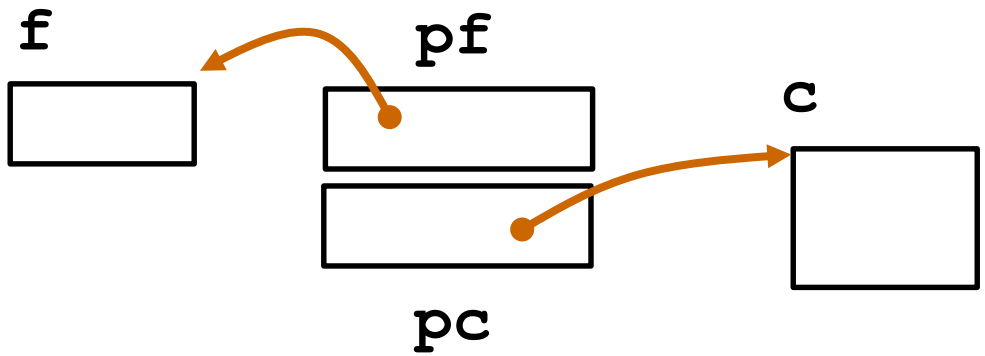
**Question:** l'occupation mémoire d'un pointeur sur une instance de la classe de base est-elle différente de celle d'un pointeur sur une classe dérivée ?

**Réponse:** un pointeur mémorise seulement **l'adresse** d'un octet en mémoire.




```
Forme    f;
Cercle   c;
Forme*   pf(&f);
Cercle*  pc(&c);
```

Sur une machine 64 bits, un pointeur occupe un mot mémoire **=> 8 octets** **quel que soit l'objet pointé**.



# Le polymorphisme d'inclusion en bref

Contexte: mécanisme associé à une hiérarchie de classes

Objectif: une reference ou un **pointeur** d'une **classe de base** donne accès **aux traitements des classes dérivées**

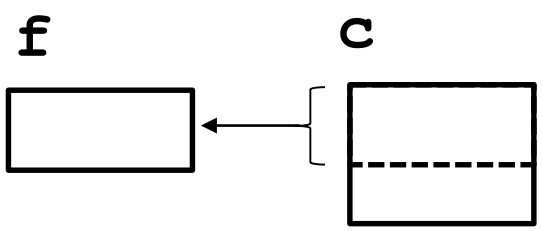
Remarque: dans la pratique le pointeur est plus utile qu'une référence

# Le pointeur = une clef pour la résolution dynamique (2)

**Question:** pourquoi le polymorphisme est-il mis en oeuvre avec un pointeur et pas une instance ?

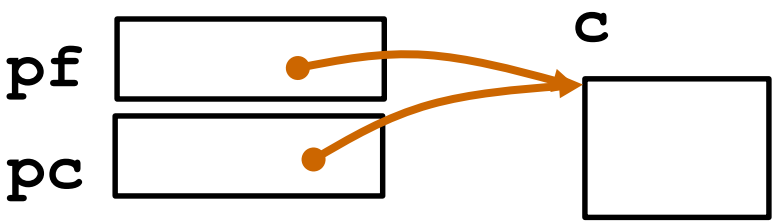
L'affectation d'une instance d'une classe dérivée à une instance d'une classe parente perd les attributs spécialisés mais aussi le lien avec la variable de la classe dérivée

```
Forme f;
Cercle c;
f = c ;
```




l'affectation d'un pointeur d'une instance d'une classe dérivée à un pointeur de la classe parente **conserve** le lien avec la variable de la classe dérivée ( => l'adresse reste la même)

```
Forme* pf(nullptr);
Cercle* pc(&c);
pf = pc ;
```



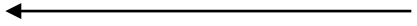
# Pourquoi une résolution «**dynamique**» et pas «**statique**» ?

La détermination de la méthode dérivée appelée à partir d'un **pointeur de la classe de base** ne se fait plus à la compilation (statiquement) mais à l'execution (dynamiquement).

```
char c;
std::cin >> c;

Forme *pf(nullptr);
if (c == 'd') pf = new Cercle;
else          pf = new Forme;

pf->description()
```



**Si on se place dans un scénario de polymorphisme**, il est impossible pour le compilateur de savoir quelle méthode `description()` doit être appelée au moment de la compilation

# Le polymorphisme est obtenu avec `virtual`

Par défaut c'est la résolution **statique** qui prime (basée sur le **type**)

La résolution **dynamique** est obtenue pour les methodes `virtual`

Comment ? En redéfinissant la méthode *(substitution)* dans les classes dérivées pour lesquelles cela est necessaire :

Il n'est pas obligatoire mais recommandé de re-indiquer `virtual`

pour informer les personnes qui utilisent cette classe dérivée

Il n'est pas obligatoire mais recommandé d'ajouter `override`

Demande au compilateur de **verifier** qu'il y a bien une méthode `virtual` de même prototype dans une classe parente ; sinon une faute de frappe pourrait conduire à la creation d'une nouvelle méthode complètement indépendante au lieu d'établir un lien avec la méthode de la classe parente.

# Le destructeur doit-il être `virtual`?

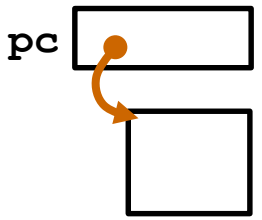
**S'il ne l'est pas** : resolution **statique** du destructeur => classe de base seulement

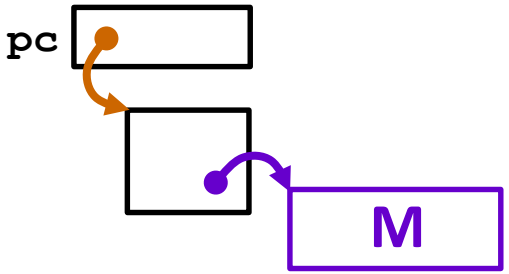
Toute hiérarchie de classe impliquant de *l'allocation dynamique dans les classes dérivées* **doit** avoir un destructeur `virtual` pour déléguer la libération correcte de la mémoire dans les classes dérivées.

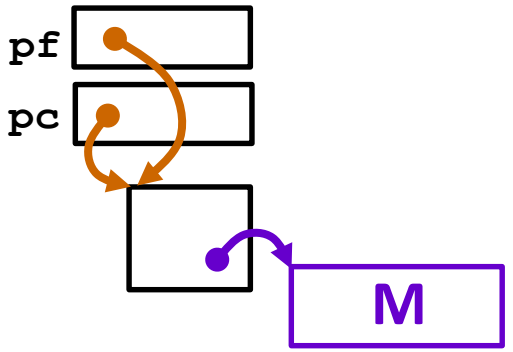
Pour les besoins de ce slide, supposons que la méthode **alloc**() réalise une allocation dynamique **d'un bloc de mémoire M** dont l'adresse est mémorisée dans un attribut.

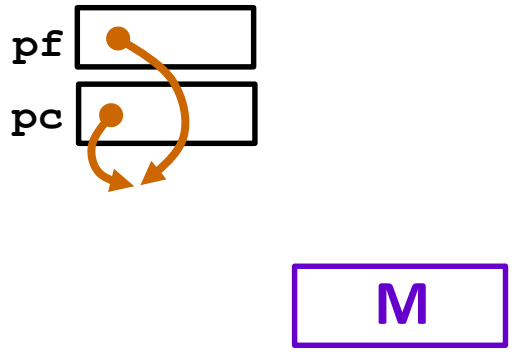
```
Cercle* pc(new Cercle);
pc->alloc();
Forme* pf(pc);
delete pf;
```

**Si** le destructeur de Forme n'est **PAS** `virtual` alors le destructeur de Cercle n'est PAS appelé et donc **le bloc M** n'est pas libéré => fuite de mémoire. Le bloc mémoire alloué pour le Cercle pourrait être entièrement libéré ; *cependant le standard du C++ mentionne un comportement indéterminé pour ce cas de figure.*



# Comment mettre en oeuvre une collection hétérogène ?

Avec un vector de pointeurs sur le type de base

**`vector<Forme*> tab;`**

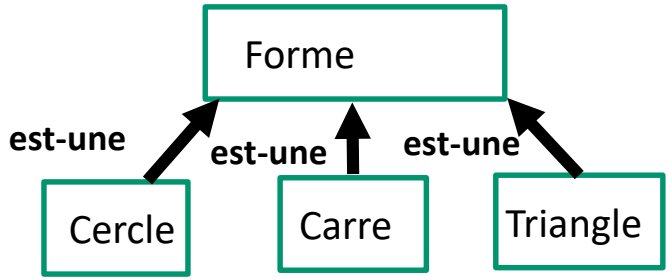
```
int main()
{
  vector<Forme*> tab;

  // ajout de l'adresse d'éléments
  // alloués dynamiquement dans tab
  tab.push_back(new Cercle);
  tab.push_back(new Triangle);
  …
  for( auto f : tab)
      f->dessin();

 // liberation tout aussi facile
 for( auto f : tab)
      delete f;
};
```

Ou un vector de **unique_ptr** sur le type de base

**`vector< unique_ptr<Forme>>`**




**delete** sur un pointeur Forme* libère le bloc mémoire alloué pour le type dérivé (si le destructeur est **virtuel**).

# Points importants

l'héritage sans polymorphisme rend peu efficace la gestion d'ensembles d'éléments appartenant à la hiérachie de classes.

Le polymorphisme permet de gérer des ensembles hétérogènes de manière transparente.

La conception d'une hiérarchie de classe doit identifier les entités/concepts les plus généraux pour la classe de base, sans qu'il soit requis de pouvoir créer des instances de cette classe (notion de *classe abstraite*).