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 qu’on veut 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

EPFL

Ce qu'on veut éviter: Quand un programme doit manipuler des ensembles de données variées (*ex: formes géométriques*), il est tentant de définir une seule classe dont un attribut categorie préciserait la nature d'une donnée (ex: sous forme d'un enum CERCLE, CARRE, TRIANGLE etc…). Ensuite on doit écrire du code qui teste cet attribut categorie (ex: avec switch) pour décider quelle fonction/méthode appeler avec quel sous-ensemble de paramètres (ex: stockés dans un vector de double ou autre) .

Pourquoi est-ce une mauvaise pratique? Pour chaque action de la classe (ex: calculs surface, perimetre, dessin,…) on va exploiter un switch sur l'attribut categorie, ce qui rend ce code dificile à maintenir sans introduire d'erreur. De plus, si on veut ajouter une nouvelle catégorie traitée par cette classe il faut avoir accès au code source de toutes les actions pour y ajouter un nouveau cas. Le code de base n'est pas protégé contre l'introduction d'erreurs à chaque extension.

N'est-ce pas déjà résolu avec une hiérarchie de classe? Presque… La hiérachie de classe assure la fiabilité lors de l'extension par dérivation. Mais c'est au prix d'introduire des nouveaux types dérivés. Là où auparavant on pouvant définir un ensemble (ex: vector) homogène (une seule classe= un seul type) on en a maintenant plusieurs (types hétérogènes).

Ce qu’on veut éviter

```
Enum Category {CERCLE, CARRE,…};
…
class SingleClassShape //single type
{
public:
   SingleClassShape();
   void dessin();
   …
private:
   Category category;
   vector<double> param;
   …
};
```

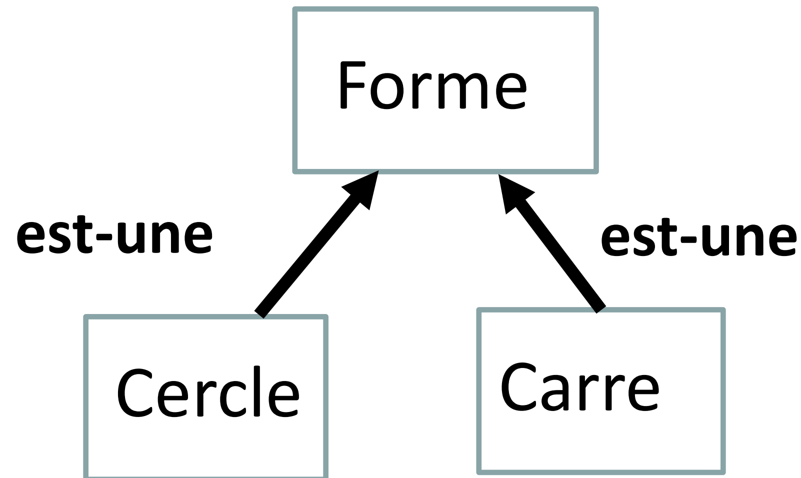
```
void SingleClassShape::dessin()
{
   switch(category)
   {
   case CERCLE:
         dessin_cercle(param); break;
   case CARRE:
         …
```

… qui pourtant est si pratique car on peut les gérer dans un unique ensemble homogène:

main.cc

```
#include "singleclassshape.h"
…
int main()
{
   vector<SingleClassShape> tab;
   …
   // ajout d’éléments dans tab
   …
   for( auto s : tab)
      s.dessin();
   …
}
```

L’héritage est un bon début



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

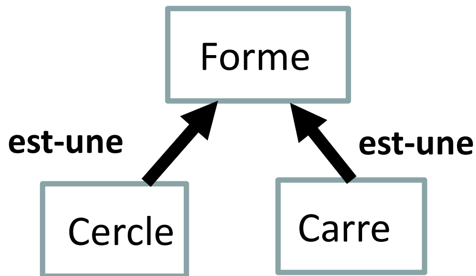
`Forme, Cercle, Carre…`

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**) ou, implicitement, celles de la superclasse s’il n’y a pas eu de redéfinition.

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'adresse d'une Forme ou d'un Cercle selon les instructions,

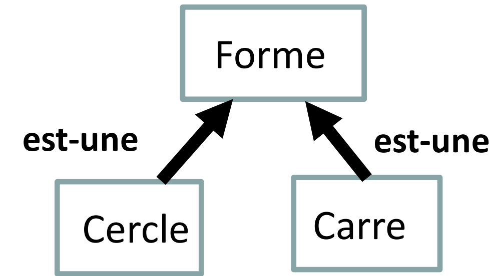
MAIS dans tous les cas, le compilateur ne voit qu'un type Forme* et il établit toujours le lien vers la méthode de la classe Forme.

Héritage: 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.**

L'occupation mémoire d'une instance dérivée est généralement plus grande ou égale à celle d'une instance de la superclasse

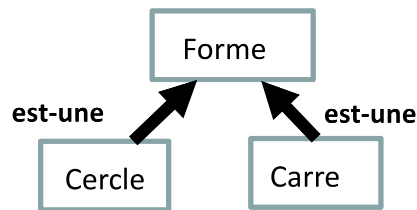



```
Forme f;
Cercle c;
f = c ;
```

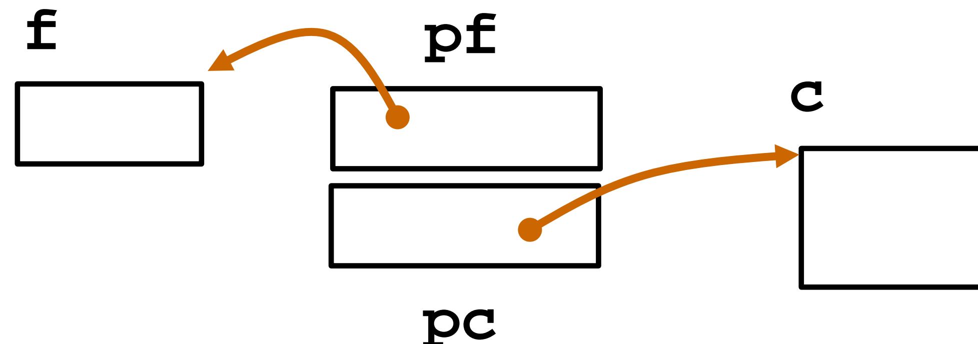
De ce fait l'affectation d'une instance d'une classe dérivée à une instance d'une classe parente se traduit par la **perte des attributs traduisant la spécialisation de la classe dérivée.**

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 ?




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



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**

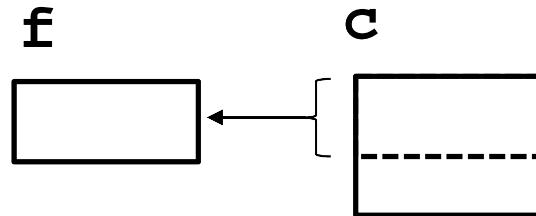
Remarque: dans la pratique c’est le pointeur le plus utile
Question: pourquoi un pointeur et pas une instance ?

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 ?

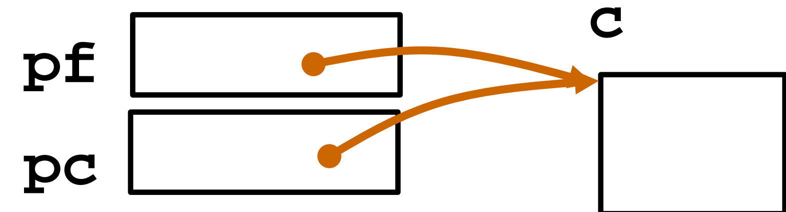
l'affectation d'une instance d'une classe dérivée à une instance d'une classe parente perd les attributs spécialisés mais aussi le lien avec la variable de la classe dérivée

```
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`

Si la méthode est redéfinie (masquage) dans une classe dérivée

Pas obligatoire mais recommandé de re-indiquer `virtual`

pour informer les personnes qui utilisent cette classe dérivée

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 *variante* de la méthode (**surcharge**) au lieu d'établir un lien avec la méthode de la classe parente.

Le destructeur doit-il être `virtual`?

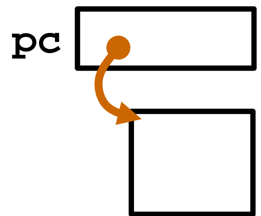
<u>S'il ne l'est pas</u> : resolution **statique** du destructeur => classe de base seulement

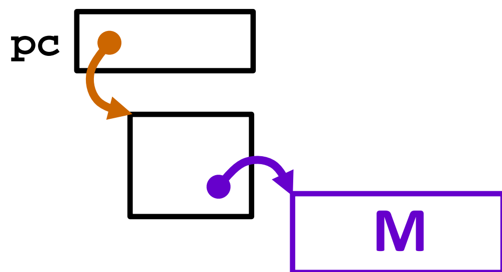
Toute hiérarchie de classe impliquant de *l'allocation dynamique dans les classes dérivées* **doit** avoir un destructeur `virtual` pour déléguer la libération correcte de la mémoire dans les classes dérivées.

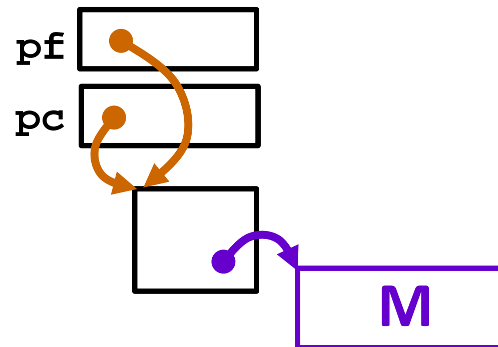
Pour les besoins de ce slide, supposons que la méthode **alloc**() réalise une allocation dynamique **d'un bloc de mémoire M** dont l'adresse est mémorisée dans un attribut.

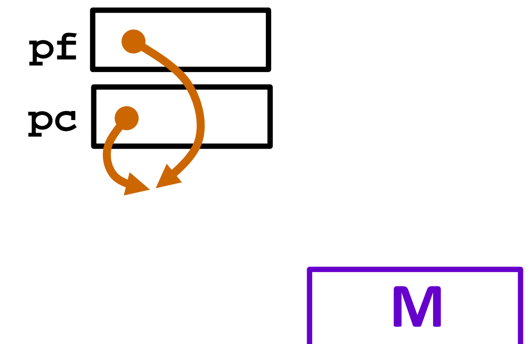
```
Cercle* pc(new Cercle);
pc->alloc();
Forme* pf(pc);
delete pf;
```

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. Par contre le bloc mémoire alloué pour le Cercle est libéré.









Comment mettre en oeuvre une collection hétérogène ?

Avec un vector de pointeurs sur le type de base

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

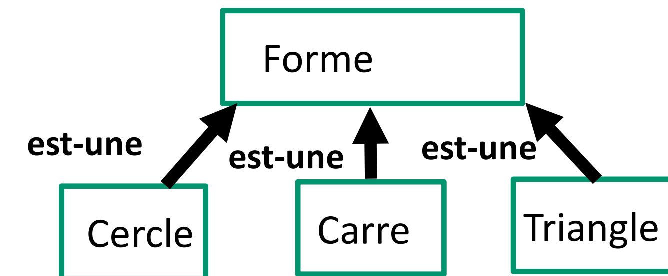
```
int main()
{
  vector<Forme*> tab;

  // 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.
Comment est-ce techniquement possible ?

Points importants

l’héritage sans polymorphisme est inefficace voire risqué.

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*).