Programmation Orientée Objet : Polymorphisme, 2e partie

Jean-Cédric Chappelier

Faculté I&C

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	19.02.26		0	Intro + compil. séparée	
2	26.02.26	1. Intro POO	0	Intro POO	
3	05.03.26	2. Constructeurs/Dest	0	Constructeurs	
4	12.03.26	3. Surcharge des opé	0	Surcharge	
5	19.03.26	4. Héritage	0	Héritage	
6	26.03.26	5. Polymorphisme	0	Polymorphisme 1	
7	02.04.26		1	Polymorphisme 2 / Collections hétérogènes	
-	09.04.26		-	vacances Pâques	
8	16.04.26	6. Héritage multiple	1	Héritage multiple	
9	23.04.26		-	-	Midtem
10	30.04.26	(7. Etude de cas)	-	Templates	
11	07.05.26		-	Structure de données abstraites ; Bibliothèques	
12	14.05.26		-	(Ascension)	
13	21.05.26	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
14	28.05.26		-	-	Examen

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Objectifs de la leçon d'aujourd'hui

- Concepts fondamentaux
- Étude de cas

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Concepts fondamentaux

Comme la semaine passée :

- notion de polymorphisme (☞ jamais « tests de type »)
- `virtual` + pointeur/référence
- méthodes virtuelles *pures* et classes *abstraites*

Application typique : **collections hétérogènes** :

- nécessité de *pointeurs* ☞ lesquels ?
- quelle interface ?
- *copie polymorphique* (complément de cours)

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Virtualité des destructeurs

...mais avant ça une question :

Destructeurs : virtuels ou non ? **Pourquoi ?**

☞ Pour garantir une *destruction polymorphique*, **lorsque cela est nécessaire** (typiquement gestion de ressources par des sous-classes polymorphiques) : quand :

- on est une super-classe (c'est-à-dire : on a au moins une sous-classe)
- on a au moins une méthode virtuelle
- on peut supposer qu'au moins une sous-classe aura à gérer des ressources (comment être sûr du contraire ?)

Mais alors...

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Virtualité des destructeurs

Mais alors... (règle des trois/cinq) :

```
virtual ~Classe() = default;

/* Remettre (ou supprimer) la copie car
 * la définition implicite du constructeur de copie est « deprecated »
 * depuis C++11 si il y a un « user-declared destructor ».
 */
Classe(const Classe&)            = default; // ou deleted
Classe& operator=(const Classe&) = default; // ou deleted

/* Remettre (ou supprimer) le déplacement car
 * si l'un des trois parmi destructeur, constructeur de copie ou affectation
 * (par copie) est défini dans la classe, alors le constructeur de déplacement
 * et l'opérateur d'affectation par déplacement implicites sont supprimés.
 */
Classe(Classe&&)            = default;      // ou deleted
Classe& operator=(Classe&&) = default;      // ou deleted
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Etude de cas

Retour et *continuation* de l'exemple des dessins (2 derniers transparents du MOOC) :

considérons un programme de modélisation graphique manipulant des *dessins*, lesquels sont des ensembles de *figures* géométriques.

Classiquement on aura :

- `Figure` comme classe abstraite, avec différentes sous-classes concrètes (cercles, rectangles, carrés, ...)
- `Dessin` comme **collection hétérogène** de figures.

Les figures étant abstraites, on a donc bien une *collection hétérogène* et donc pas le choix pour le contenu du tableau dynamique : pour avoir du *polymorphisme*, il *faut* des **pointeurs**.

La question est : quels pointeurs ? et comment les gérer ?

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Conception

Questions à se poser :

- qui est « *propriétaire* » du contenu ?
- quel est le *statut du contenu* : **personnel** ou **partagé** ?

En clair, une figure dans un dessin donné est elle *universelle* (partagée par d'autres dessins) ou *spécifique* à ce dessin précis ?

Par exemple, si je colorie en rouge le cercle 23 du dessin 18, est-ce que seul ce cercle sera rouge ou bien d'autres ?
du même dessin ? d'autres dessins ?

Les réponses à ces questions dépendent du cadre général du programme et de sa **conception**, et n'ont pas de réponse unique.

Dans le cadre choisi ici (dessins), il semble naturel que les éléments de la collection (les figures) soient personnels et uniques (= pas même de partage interne).

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Interface ?

Autre question :

doit-on ou non montrer les pointeurs (à l'extérieur) ?

- OUI,

```
dessin.ajoute(&cercle23);
dessin.ajoute(new Cercle(5.41));
```

- ou NON ?

```
dessin.ajoute(cercle);
dessin.ajoute(Cercle(5.41));
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Montrer les pointeurs

Si *oui*, alors on aura des prototypes du genre

```
void Dessin::ajoute(Figure* nouvelle);
```

Attention ! MAIS cela supposera :
(il faudra que l'utilisateur de la classe `Dessin` y fasse bien attention !
☞ risques de mauvaise programmation)

① que l'on ajoute à chaque fois une *nouvelle* instance de figure ; typiquement :
```
mondessin.ajoute(new Cercle(4.86));
```
ou
```
mondessin.ajoute(&c23); // attention à ne pas l'ajouter ailleurs !!
```

② que le propriétaire (nous ou l'utilisateur de la classe, en fonction de la réponse à la première question) s'occupe de la gestion mémoire
(ne pas oublier les `delete`, ne pas les faire trop tôt non plus).

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Cacher les pointeurs

Si *non*, alors on aura des prototypes du genre

```
void Dessin::ajoute(Figure const& nouvelle);
```

MAIS cela supposera :

1. que l'on s'occupe (nous) de la gestion mémoire (ce qui est plutôt bien car plus localisé, donc moins de risque d'erreur) ;
2. que l'on fasse à chaque fois une copie de la figure géométrique reçue comme exemple (puisque l'on est dans le cadre « unique et personnel ») ; typiquement avoir une *copie profonde polymorphique* au niveau des `Figure`s.

```
class Figure { ...
   virtual Figure* copie() const = 0;
    ... };
class Cercle : public Figure { ...
   virtual Cercle* copie() const override {
     return new Cercle(*this);
   }
  ... };
```

On parle de « retour *covariant* » (*covariant return*)

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Quels pointeurs ?

Autre question, en C++11 : quel type pointeur ?

RAPPEL :

utilisation	sur des données	sur des fonctions
référence	références (ou pointeurs à la C)	nom de la fonction
généricité	pointeur à la C ou index dans un tableau	`std::function` ou pointeur à la C
allocation dynamique	smart-pointers surtout `unique_ptr` (ou pointeurs à la C)	—

« *Utilisez des références quand vous pouvez, des pointeurs quand vous devez.* »

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Quels pointeurs ?

Autre question, en C++11 : quel type pointeur ?

- « à la C » (`Figure*`)
  ☞ gestion « à la main » de la désallocation mémoire

- `unique_ptr`
  ☞ recommandé, mais attention à l'aspect « unique ».
  Exemple : parcours de la collection :

```
for (auto & el : collection)
```

ou

```
for (auto const& el : collection)
```

Par ailleurs les `unique_ptr` ne peuvent pas être covariants :-(
Mais on peut utiliser le truc suivant :

```
unique_ptr<Cercle> Cercle::cloneMe() const
{ return unique_ptr<Cercle>(new Cercle(*this)); }
virtual unique_ptr<Figure> Cercle::copie() const
{ return cloneMe(); }
```

- [hors cours] `shared_ptr` ☞ déconseillé ici et attention aux pièges

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Héritage ou encapsulation de `vector` ?

Dernière question :
le dessin **est**-il ou bien **a**/**possède**-t-il un tableau dynamique de figures ?

☞ Nous vous conseillons ici de choisir l'*encapsulation*
(le dessin possède un tableau de figures).

Par exemple :

```
class Dessin {
  ...
  private:
    vector<unique_ptr<Figure>> contenu;
};
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Pour résumer, version 1/3

Voici *un* exemple possible avec `unique_ptr` **et copie polymorphique** :

utilisation : `dessin.ajoute(c);` (pour **un** `Cercle c;` existant)

```
class Dessin
{
public:
  void ajoute(Figure const& objet) {
    contenu.push_back(objet.copie());
  }
private:
  vector<unique_ptr<Figure>> contenu;
};
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Pour résumer, version 1/3 (suite)

```
class Figure {
  // ...
  virtual unique_ptr<Figure> copie() const = 0;
  // ...
};

class Cercle {
  // ...
  unique_ptr<Cercle> Cercle::clone() const
  { return unique_ptr<Cercle>(new Cercle(*this)); }

  // voire une méthode template.... (plus tard dans le cours)
  virtual unique_ptr<Figure> Cercle::copie() const
  { return clone(); }
  // ...
};
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Pour résumer, version 2/3 a

Voici *un* exemple possible avec `unique_ptr` **sans** copie polymorphique :

utilisation : `dessin.ajoute(new Cercle());`

```
class Dessin
{
public:
  void ajoute(Figure* objet) {
    if (objet != nullptr) {
      contenu.push_back(unique_ptr<Figure>(objet));
    }
  }
private:
  vector<unique_ptr<Figure>> contenu;
};
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Pour résumer, version 2/3 b

Voici un *autre* exemple possible avec `unique_ptr` **sans** copie polymorphique :

utilisation : `dessin.ajoute(make_unique<Cercle>());`

```
class Dessin
{
public:
  void ajoute(unique_ptr<Figure> && objet) {
    contenu.push_back(move(objet));
  }
private:
  vector<unique_ptr<Figure>> contenu;
};
```

Note : le passage par rvalue-reference (le `&&` de `ajoute()`) n'est pas nécessaire ici puisque les `unique_ptr` ne sont pas copiables, mais est préférable dans le cas général (p.ex. avec des `shared_ptr`)

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Pour résumer, version 3/3

Voici *un* exemple possible avec « **pointeurs à la C** » (**sans** copie polymorphique) :

utilisation : `dessin.ajoute(new Cercle());` ou `dessin.ajoute(&cercle21);`

```
class Dessin
{
public:
  void ajoute(Figure* objet) {
    if (objet != nullptr) {
      contenu.push_back(objet);
    }
  }
  ~Dessin() { for (auto o : contenu) delete(o); }
  // interdire la copie
  Dessin& operator=(const Dessin&) = delete;
  Dessin(const Dessin&)            = delete;
  Dessin() = default; // remettre le cteur par défaut
private:
  vector<Figure*> contenu;
};
```

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam

Synthèse

	version 1/3	version 2a/3	version 2b/3	version 3/3
interface :	`ajoute(Classe const& )`	`ajoute(Classe*)`	`ajoute(unique_ptr<Classe> &&)`	`ajoute(Classe*)`
collection :	`unique_ptr<Classe>`	`unique_ptr<Classe>`	`unique_ptr<Classe>`	`Classe*`

Au niveau de ce cours, je vous conseille (hors exercices courts et faciles) :

- Tant que faire se peut de donner la propriété à la collection
- et dans ce cas d'utiliser des `unique_ptr` en C++11 ou des pointeurs à la C, gérés en interne de la collection
  (copie polymorphique [« version 1/3 »] ou déplacement [« version 2b/3 »])
- sinon (si vous ne pouvez pas donner la propriété à la collection ou ne vous sentez pas le niveau de faire de la copie polymorphique), de transférer des adresses d'objets (extérieurs) existants *au moins aussi longtemps* que la collection elle-même.
  Il faudra donc **le garantir !**
  [« version 3/3 »]

©EPFL 2026
Jean-Cédric Chappelier
& Jamila Sam