

Programmation Orientée Objet : Polymorphisme, 2^e partie

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Organisation du travail (semestre)

	MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
24.02.22		0	Intro + compil. séparée	
03.03.22	1. Intro POO	0	Intro POO	
10.03.22	2. Constructeurs/Dé	0	Constructeurs	
17.03.22	3. Surcharge des op	0	Surcharge	
24.03.22	4. Héritage	0	Héritage	
31.03.22	5. Polymorphisme	0	Polymorphisme 1	
07.04.22		1	Polymorphisme 2 / Collections hétérogènes	
14.04.22	6. Héritage multiple	1	Héritage multiple	
21.04.22		-	vacances Pâques	
28.04.22	(7. Etude de cas)	-	Templates	Série notée
05.05.22		-	Bibliothèques 1 (SDA)	
12.05.22		-	Bibliothèques 2	
19.05.22	(7. Etude de cas)	-	Révisions	
26.05.22		-	(Ascension)	
02.06.22		-	-	Examen

Objectifs de la leçon d'aujourd'hui

- ▶ Concepts fondamentaux
- ▶ Étude de cas

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** :

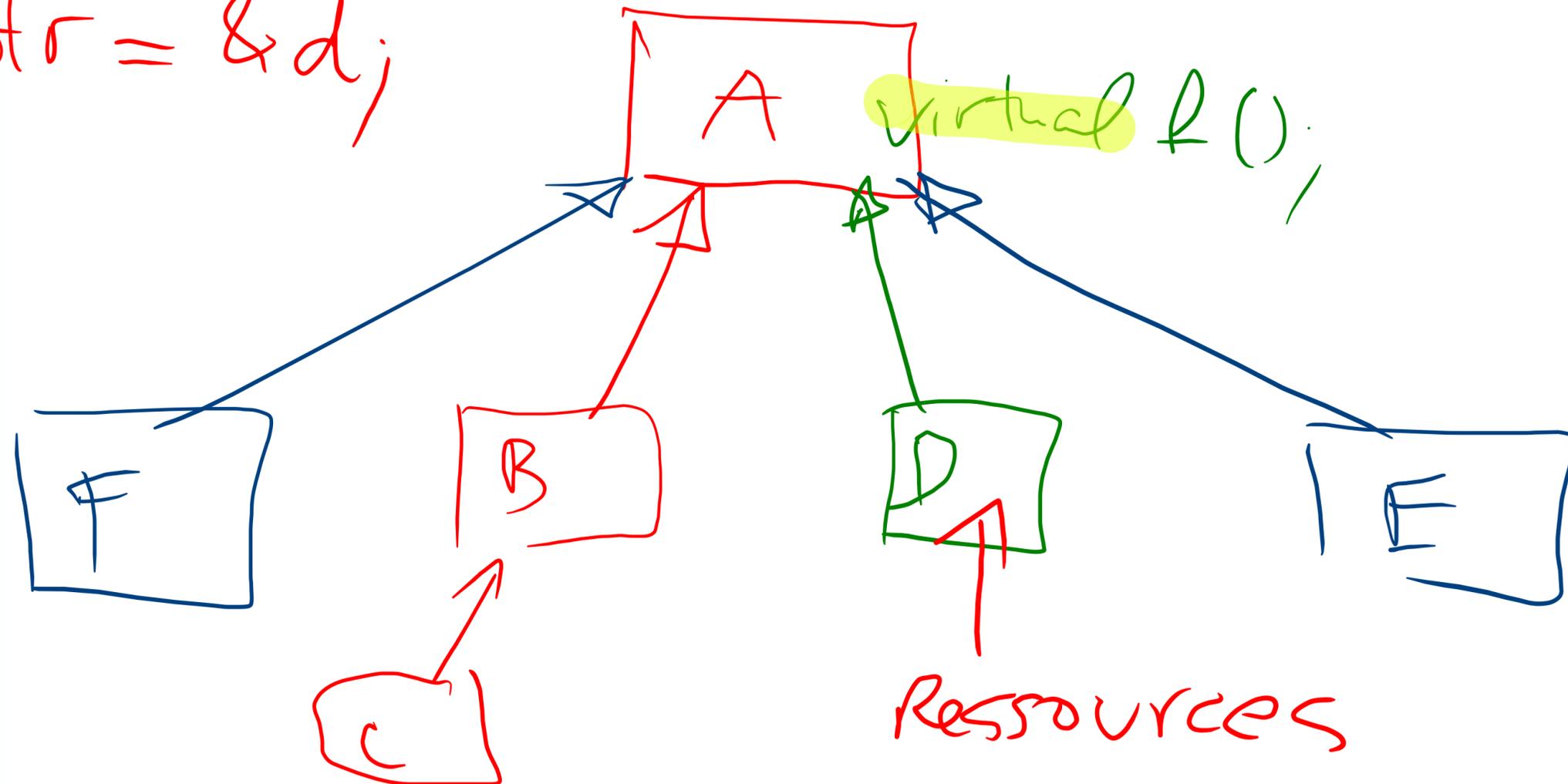
- ▶ nécessité de *pointeurs* 👉 lesquels ?
- ▶ quelle interface ?
- ▶ *copie polymorphique* (complément de cours)

Virtualité des destructeurs

...mais avant ça une question :

Destructeurs : virtuels ou non ? Pourquoi ?

$A * ptr = \&d;$



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)

Virtualité des destructeurs

...mais avant ça une question :

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

A
virtual \neq

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

 Mais alors...

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
```

Etude de cas

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

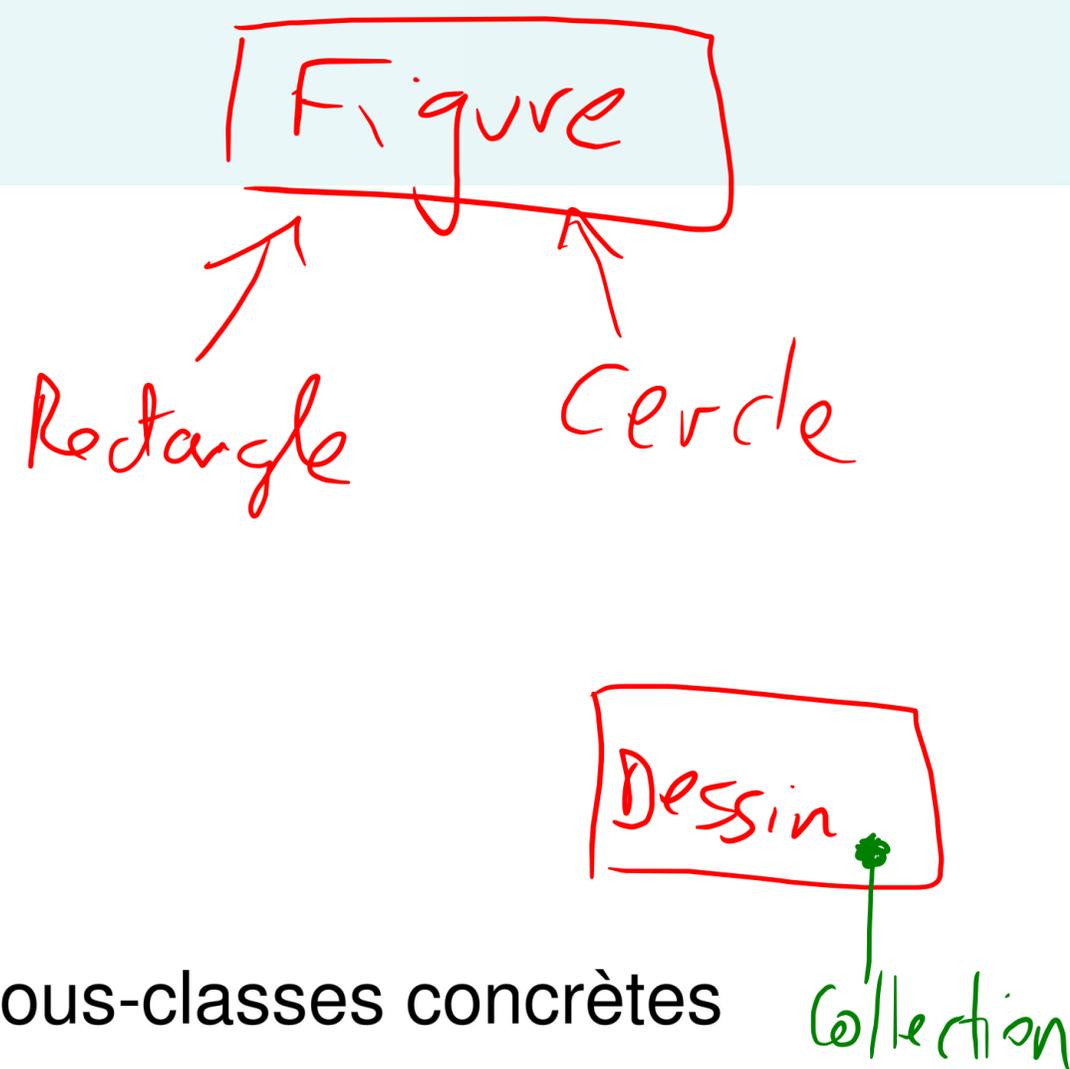
considérons un programme de modélisation graphique manipulant des *dessins*, lesquels sont des ensembles de *figures* géométriques.

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 ?



Conception



Questions à se poser :

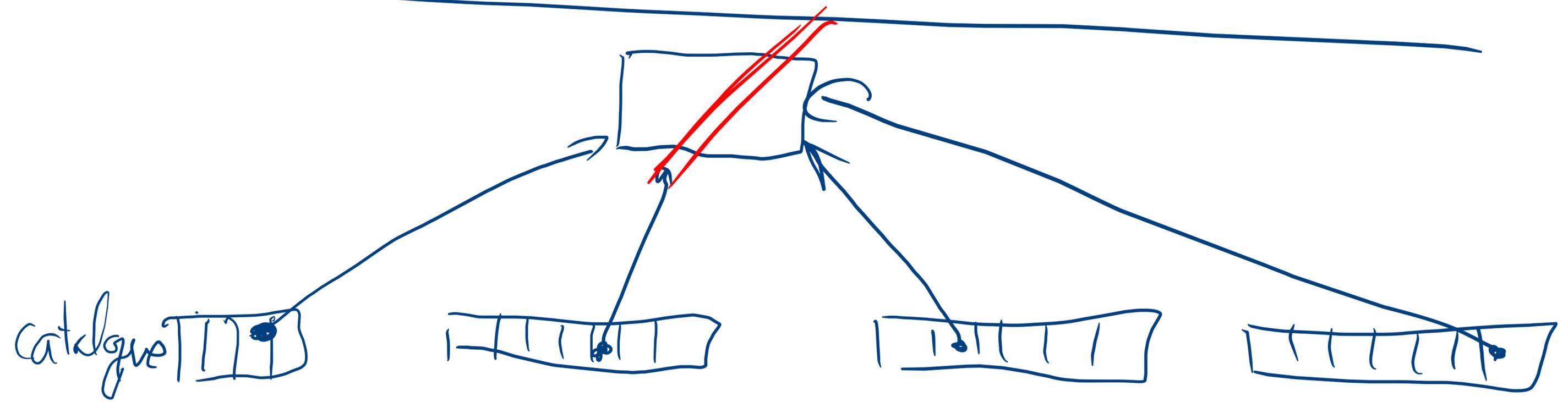
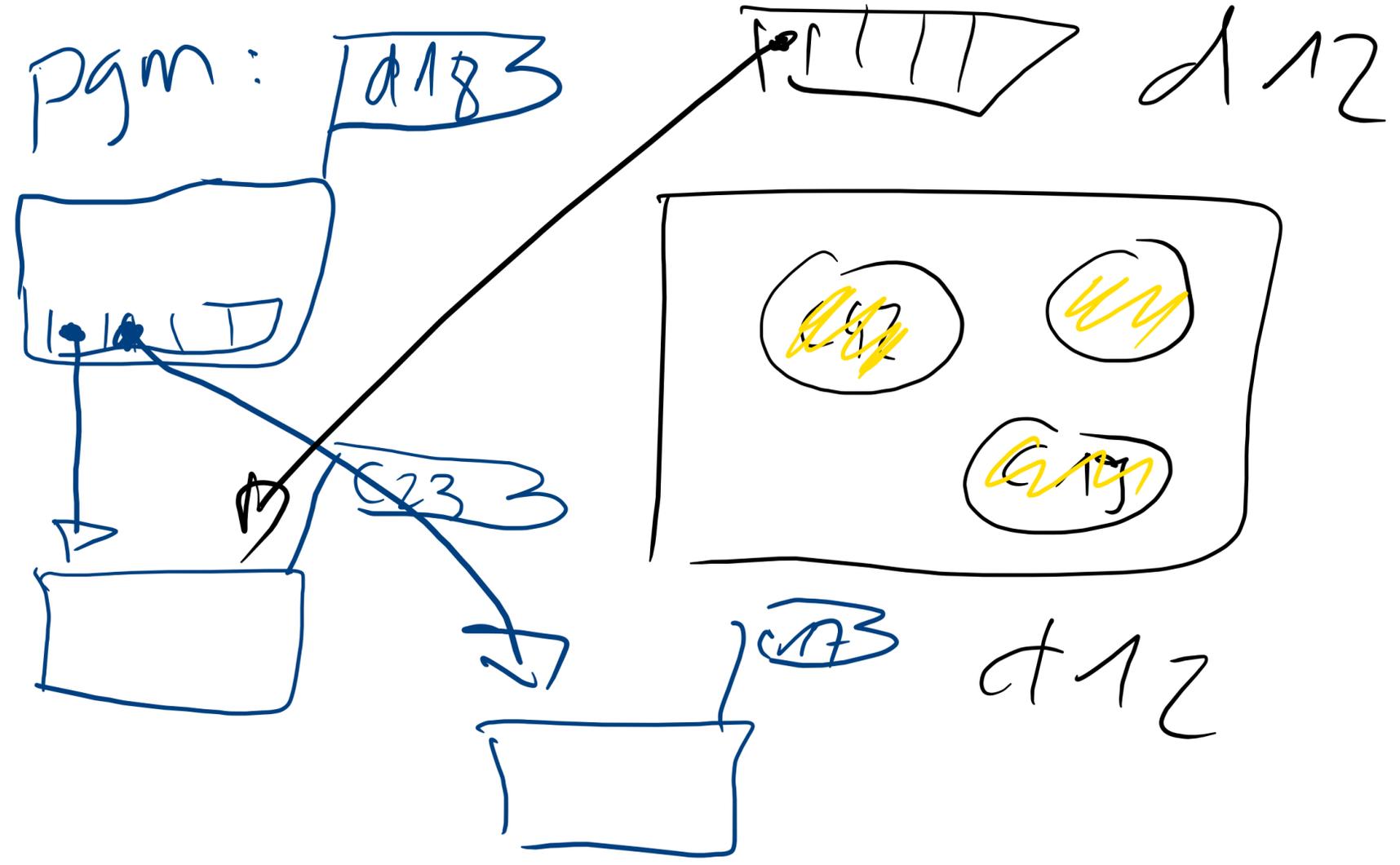
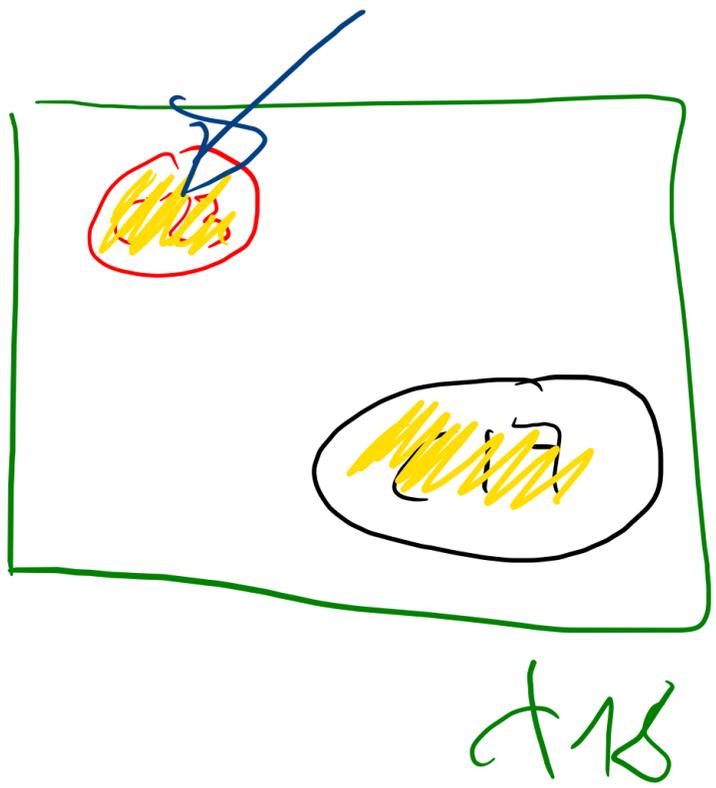
- ▶ qui est « *propriétaire* » du contenu ?
- ▶ quel est le *statut du contenu* : **personnel** ou **partagé** ?

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 uniques et personnels.



Interface ?

Autre question :

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

► OUI,

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

ajoute (Figure *)

► ou NON ?

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

ajoute (Figure
const &)

Montrer les pointeurs

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

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

MAIS cela supposera :

(il faudra que l'utilisateur de la classe `Dessin` y fasse bien attention !

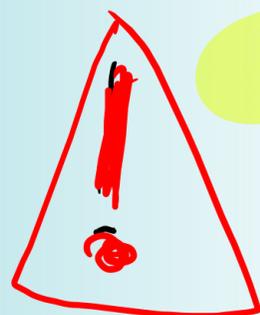
☞ risques de mauvaise programmation)

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).

Cacher les pointeurs

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

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

ajoute (Rectangle(...))

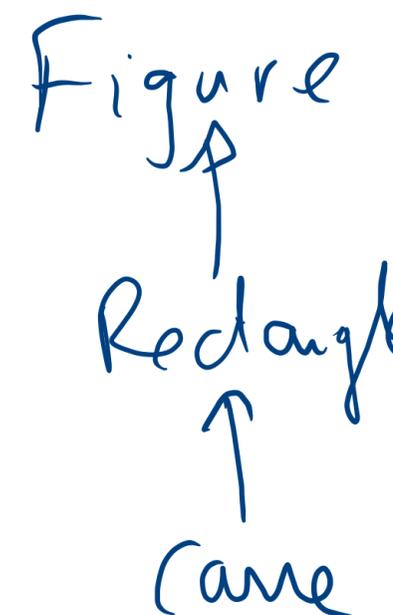
MAIS cela supposera :

→ Dessin

- 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 **Figures**.

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

Figure



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

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)
```

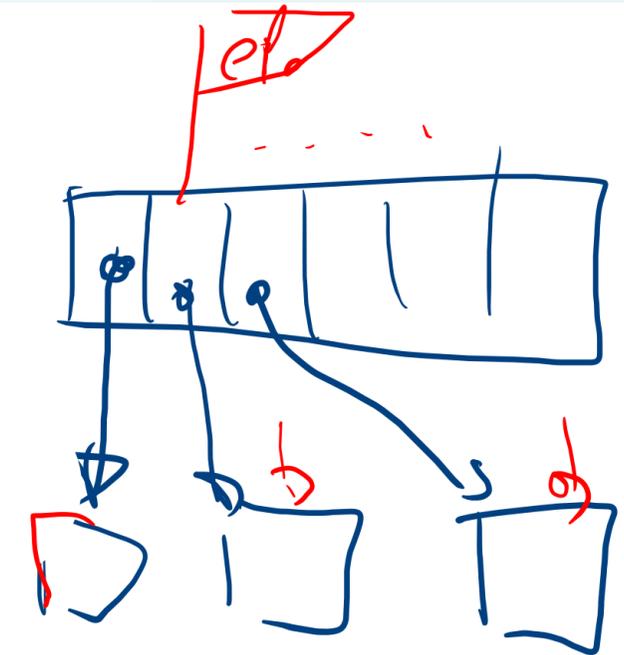
🌀🌀 Par ailleurs les **unique_ptr** ne peuvent pas être covariants : - (

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



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;  
};
```

Pour résumer, version 1/3

ne pas montrer les ptr } difficile
sûr

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;
};
```

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(); }
    // ...
};
```

Pour résumer, version 2/3

ptr dans l'interface | *facile*

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

plus risqué

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;
};
```

Pour résumer, version 3/3

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

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

```
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;
};
```

Synthèse

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, mais gérés en interne de la collection (copie polymorphique)
- ▶ 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 !)