Programmation Orientée Objet (C++) : Synthèse des concepts de l'orienté objets

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Objectifs du cours d'aujourd'hui

L'objectif de ces quelques transparents est de vous rafraîchir la mémoire en rappelant les principaux points.

Vous avez abordé jusqu'ici :

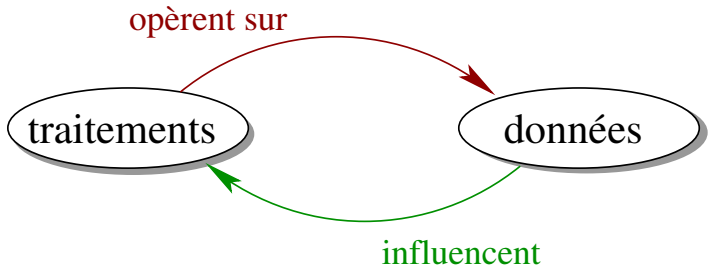
1. les bases de la programmation procédurale ;
2. les bases de la programmation orientée objets.

☞ Il nous reste à aborder quelques thèmes d'approfondissement : structures de données et «templates» ainsi qu'un survol de la librairie standard

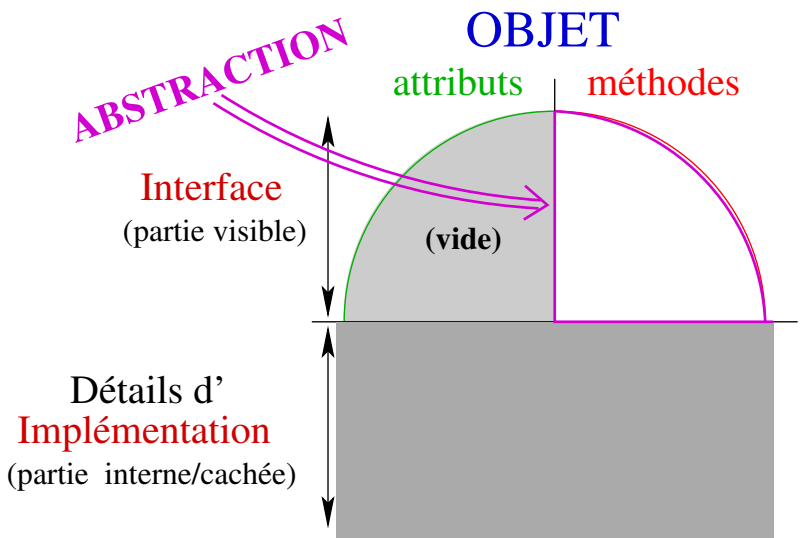
Jamila Sam
& Jean-Cédric Chappelier

Qu'avons nous vu en programmation ?

programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**



en programmation orientée objets, on regroupe dans le même objet **les traitements et les données** qui lui sont spécifiques (principe d'**encapsulation**)



Qu'avons nous vu en programmation ?

programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**

Algorithme	S.D.A.
Traitements	Données
Expressions & Opérateurs Structures de contrôle Fonctions	Variables Portée Chaînes de caractères Tableaux statiques Tableaux dynamiques Structures Pointeurs
Entrées/Sorties	

Jamila Sam
& Jean-Cédric Chappelier

Qu'avons nous vu en programmation ?

en programmation orientée objets, on regroupe dans le même objet **les traitements et les données** qui lui sont spécifiques (principe d'**encapsulation**)

<table>
<tr><th colspan="2">Objet</th></tr>
<tr><td colspan="2">Encapsulation et Abstraction
Classes
Héritage simple/multiple
Polymorphisme
Classes abstraites/virtuelles
Résolution des collisions de noms</td></tr>
<tr><th>Traitements</th><th>Données</th></tr>
<tr><td>Méthodes
Constructeurs & Destructeurs
Const
Virtuelles (pures)
Surcharge d'opérateurs(interne/externe)</td><td>Attributs
Appels aux constructeurs des attributs (hérités)
Statiques</td></tr>
<tr><td colspan="2">Privés/protégés/publiques
Hérités/cachés (: :)</td></tr>
</table>

FONDAMENTAUX

1. **déclarez avant d'utiliser**
   - variables
     ```
     int i;
     vector<double> v;
     ```
   - fonctions ☞ prototype
     ```
     double sin(double x);
     bool cherche_valeur(Listechainee l, Valeur v);
     ```
   - classes ☞ Attributs et prototypes des méthodes
2. **modularisez** / décomposez / pensez « atomique » et « **objet** »
   - 2.1 conception (qu'est ce qu'on veut ?)
   - 2.2 implémentation (comment ça se réalise ?)
   - 2.3 syntaxe (comment ça s'écrit ?)
   - 2.4 tests (où sont mes fautes, comment pourrais-je les tester ?)

« fondamentaux » de la POO

1. **encapsulation** Objet = attributs + méthodes

```
class Rectangle {
public:
  double surface() { ... };
  ...
private:
  double hauteur;
  double largeur;
}
```

Attributs et méthodes publiques ☞ Interface de la classe (abstraction)

2. **héritage**

```
class RectangleColore : public Rectangle {
   Couleur couleur;
   //...};
```

3. **polymorphisme** le choix du type se fait à l'exécution, en fonction de la nature réelles des instances (typage dynamique)
Ingrédients : Pointeurs/Références + méthodes virtuelles

Pour réviser...

- prendre les tableaux synthétiques des transparents 3 et 4
- prendre les fiches résumé
- et pour chacun des points, se demander si on sait :
  - de quoi ça parle ?
  - ce que ça veut dire ?
  - l'utiliser ?

☞ se focaliser sur les **concepts**.

Les détails de syntaxe (comment ça s'écrit) peuvent être ensuite rapidement retrouvés dans la fiche résumé, si on sait ce qu'on cherche (c'est-à-dire si on a le concept)

Synthèse

- polymorphisme et collections hétérogènes
- héritage de containers (typiquement `vector`)
- méthodes `const`
- classes virtuelles

Un exemple concret

Reprenons l'un de nos exercices de séries. Il s'agit de :

définir une collection de figures géométriques

Une figure peut être un `cercle`, un `carre` ou un `triangle`

À chacune de ces formes est associée une méthode d'affichage spécifique `affiche()`

La collection sera implémentée au moyen d'une classe `Dessin`

La classe `Dessin` permettra notamment d'invoquer la bonne fonction d'affichage pour chaque figure de la collection

☞ COMMENT FAIRE ?

Pensons Objet

On nous demande :
d'implémenter une collection, Dessin, de figures géométriques

Objet `Figure`

- ☞ attributs : ? et méthodes : `affiche()`
- ☞ afficher la description d'une `Figure`

Objet `Dessin`

- ☞ attribut : une « collection » de `Figure` ? et méthodes : `affiche()`
- ☞ afficher les figures de la collection

d'où déjà :

```
class Figure {
  ...
  void affiche () const;
  ...
};

class Dessin {
  // une collection de Figures
  ...
  void affiche () const;
};
```

Affinons la description de nos classes

Commençons par la classe `Figure` :
c'est plutôt ...« abstrait » :

des attributs concrets (rayon, coté, etc.) à afficher ne sont spécifiables que pour des figures géométriques précises (`carre`, `triangle`, etc.)

de même la méthode `affiche` va dépendre du type de `Figure` considérée

Continuons donc à penser « objet » et « atomique »

- ☞ il nous faut aussi décrire des **formes spécifiques** de `Figures`

Ici des `Cercles`, des `Carres` et des `Triangles`

- ☞ Pour ces types d'objets, on sait concrètement quels sont les attributs typiques et on a (au moins) une idée de comment les afficher

Affinons la description de nos classes (2)

Exemple des `Cercles` :

```
class Cercle  {
  void affiche() const {
    cout << "Un cercle de rayon " << rayon << endl;
  }
  double rayon;
};
```

La classe Cercle

Spécifions un peu plus complètement cette classe :

Tout d'abord respectons les principes d'une bonne encapsulation :

Définissons proprement **l'interface de la classe** (i.e, ce qui doit être visible depuis l'extérieur de la classe et ce qui n'a pas besoin de l'être)

```
class Cercle  {
public:  //DROIT D'ACCES
  void affiche() const  {
    cout << "Un cercle de rayon " << rayon << endl;
  }
private: //DROIT D'ACCES
  double rayon;
};
```

La classe Cercle (2)

Il nous faut aussi des constructeurs/destructeurs

```
class Cercle  {
 public:
        Cercle(double x = 0.0) // CONSTRUCTEUR PAR DEFAUT
  : rayon(x) {
    cout << "Et hop, un cercle de plus !" << endl;
  }

  // pas vraiment necessaire, mais pour l'exemple
  Cercle(const Cercle& c)
  : rayon(c.rayon) {
    cout << "Et encore un cercle qui fait des petits !" << endl;
  }

  // pas vraiment necessaire non plus ici
  ~Cercle() { cout << "le dernier cercle ?" << endl; }

  void affiche() const {
    cout << "Un cercle de rayon " << rayon << endl;
  }

 private:
  double rayon;
};
```

La classe Cercle (3)

Ce n'est pas tout !
Nos `Cercles` sont des « spécialisations » possibles de `Figure`.
Ils seront d'ailleurs manipulés dans notre collection comme des `Figure`s quelconques (*polymorphisme*)

☞ il faut donc que `Cercle` **hérite** de `Figure` :

```
class Cercle : public Figure {
 public:
  Cercle(double x = 0.0)
  : rayon(x) {
    cout << "Et hop, un cercle de plus !" << endl;
  }
  ...
 private:
  double rayon;
};
```

On fait pareil pour toutes les autres types de `Figure`s, et on obtient ainsi les briques de base de notre implémentation

La classe Dessin

Une façon d'implémenter notre collection hétérogène consiste à doter la classe `Dessin` d'un attribut stockant (« *encapsulant* ») une liste de de `Figure`s (un tableau dynamique par exemple)

question : **Est-ce la meilleure façon de procéder ?**

- ☞ Oui, si c'est comme cela qu'on le conçoit (« un dessin **a**/**possède** une collection de figures »).

Mais une autre vision est aussi possible : « un dessin **est** une collection de figures »

- ☞ on fait alors *hériter* `Dessin` d'une classe container (`vector` par exemple)

On bénéficie alors directement (« hérite ») des méthodes existants pour les `vector` ! ! (constructeurs, destructeur, `push_back()`, `size()`, etc.)

[mais attention le destructeur de `vector` n'est pas virtuel]

- ☞ COMMENT FAIRE ?

Héritage d'un container

On veut donc faire hériter `Dessin` d'une classe vecteur de `Figures` :

```
class Dessin : public vector<Figure> { //HERITAGE DE VECTOR
public:
  ...
  void affiche() const {
    cout << "Je contiens :" << endl;
    for (auto figure : *this) {
      figure.affiche();
    }
  }
};
```

Mais regardons notre fonction `affiche` de la classe `Dessin`...

On souhaite qu'elle invoque pour chaque `figure`, la méthode d'affichage appropriée à la nature réelle de l'instance stockée.
☞ Il est temps de se préoccuper un peu de **polymorphisme** !

Pointeurs et polymorphisme

Polymorphisme : **Pointeurs/Références + méthodes virtuelles**

- ☞ notre classe `Dessin` **doit** donc manipuler des pointeurs sur les instances de `Figure`s et non pas les instances elles-mêmes !

```
class Dessin : public vector<Figure*> { // POINTEURS !!
public:
  ...
  void affiche() const {
    cout << "Je contiens :" << endl;
    for (auto figure : *this) {
      figure->affiche();
    }
  }
};
```

Le premier ingrédient est désormais fourni !
(mais attention aux classes contenant des pointeurs !)

Méthodes virtuelles

Deuxième ingrédient : la fonction `affiche` de la classe `Figure` doit être **virtuelle**

```
class Figure {
  ...
  virtual void affiche() const; // METHODE VIRTUELLE !
  ...
};
```

Ajout d'une Figure

L'énoncé de l'exercice précise que l'on veut une méthode `ajouteFigure` ajoutant à la collection la copie d'une figure passée en paramètre.

```
class Dessin : public vector<Figure*> {
public:

  void ajouteFigure(const Figure& fig) {
    push_back(fig.copie());
  }
  void affiche() const {
    cout << "Je contiens :" << endl;
    for (auto figure : *this) {
      figure->affiche();
    }
  }
};
```


Jamila Sam
& Jean-Cédric Chappelier

Ajout d'une Figure (2)

Une méthode `copie` doit donc être fournie dans la classe `Figure` :

```
class Figure {
public:
        virtual void affiche () const;
        // COPIE POLYMORPHIQUE :
        virtual Figure* copie() const;
..
};
```

`copie` doit aussi être virtuelle pour les mêmes raisons que `affiche`

La classe Figure

Les méthodes de la classe `Figure` ont ceci de particulier que l'on sait qu'elles doivent exister mais qu'on ne sait pas exactement comment les coder

```
class Figure {
public:
  virtual void affiche () const { ??? }
  virtual Figure* copie() const { ??? }
..
};
```

La classe Figure

Au lieu de donner des définitions arbitraires à `affiche` et `copie`, il faut les déclarer comme **virtuelles pures**.

Elles n'ont alors pas de définition associée et la classe `Figure` devient une **classe abstraite**

```
class Figure {
public:
        // METHODES VIRTUELLES PURES :
        virtual void affiche () const = 0;
        virtual Figure* copie() const = 0;
  ...
};
```

Ces méthodes ne seront définies que dans les sous-classes « concrètes » de `Figure`

Finalisation des classes

Terminons une première version de nos classes :

il faut fournir les définitions concrètes de `copie` dans les sous-classes dérivées de `Figures`

Exemple :

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

Finalisation des classes (2)

Il faut aussi fournir des destructeurs à toutes classes pour **désallouer** proprement la mémoire utilisée par notre collection :

```
class Dessin : public vector<Figure*> {
public:
  ~Dessin() {
    cout << "Le dessin s'efface..." << endl;
    for (auto figure : *this) delete figure;
        clear();
  }
...
};
class Figure {
public:
  ...
  virtual ~Figure() { cout << "Une figure de moins." << endl; }
};
class Cercle : public Figure {
public:
  ...
  ~Cercle() { cout << "le dernier cercle ?" << endl; }};
```

Pourquoi le destructeur de `Figure` doit-il être virtuel ? (voir cours sur le polymorphisme)

Méthodes const

Pour améliorer notre code, toutes les méthodes ne modifiant pas les attributs de leurs classes ont été déclarées comme `const`

C'est le cas de toutes les méthodes `affiche` et `copie`

Exemple :

```
class Figure {
public:
        virtual void affiche () const= 0; // METHODE CONST
  virtual Figure* copie() const = 0;
  virtual ~Figure() { cout << "Une figure de moins." << endl;
};
```

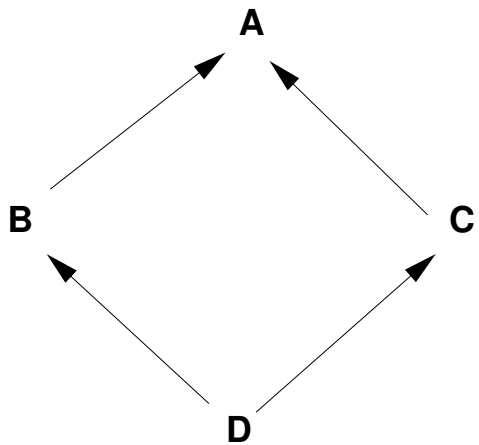
☞ Meilleures spécification des intentions du programmeur, contrôles syntaxiques supplémentaires possibles

On peut ensuite améliorer à souhait le codage de nos classes en fournissant notamment des méthodes de copie profonde et des surcharges d'opérateurs (dont =), surtout là où il y a des pointeurs (`Dessin`).

Classes virtuelles

Passons à un autre sujet délicat, lié cette fois à l'héritage multiple.

Supposons que nous ayons à coder une hiérarchie de classes « en losange » se présentant comme suit :



Chaque instance de la classe `D` hérite, *a priori*, deux fois des attributs et méthodes de `A`

Examinons ce qui se passe lorsque l'on exécute le code suivant...

La classe A

```
class A {
public:
  int a;

  A(int i) : a(i)
  { cout<< "Creation de A" << endl; }

  virtual ~A() {cout << "Destruction de A"<<endl; }

  void affiche() const{
    cout << "A: "  << a <<endl;
  }
};
```

Les classes B et C

```
class B: public  A {
public:
  int b;
  B(int i, int j)
    :A(i), b(j)
  {cout<<"Creation de B" <<endl;}
  virtual ~B(){cout << "Destruction de B" <<endl;}
  void affiche() const{
    A::affiche();
    cout << "B: " << b <<endl;
  }
};

class C: public A {
public:
  int c;
  C(int i, int j)
    :A(i), c(j)
  {cout << "Creation de C" << endl;}
  virtual ~C(){cout << "Destruction de C"<< endl;}
  void affiche() const{
    A::affiche();
    cout << "C: "  << c <<endl;
  }
};
```

La classe D

```
class D: public B, public C {
public:
  int d;

  D(int i, int j, int k, int l)
    :B(i,j), C(0,k), d(l)
  {cout << "Creation de D" << endl;}

  virtual ~D(){cout << "Destruction de D" << endl;}

  void affiche() const{
    C::affiche();
    B::affiche();
    cout<< "D: " << d << endl;
  }
};
```

Exécution

Si l'on exécute le petit main suivant :

```
int main ()
{
  D x(1,2,3,4);
  x.affiche();
  return 0;
}
```

On obtient le résultat d'exécution suivant :

```
Création de A
Création de B
Création de A
Création de C
Création de D
A: 0
C: 3
A: 1
B: 2
D: 4
Destruction de D
Destruction de C
Destruction de A
Destruction de B
Destruction de A
```

Classe A virtuelle

Pour éviter cette duplication, il faut déclarer la classe A comme **virtuelle**. Ceci se fait en déclarant le lien d'héritage de B **et** C par rapport à A comme virtuel :

```
class B: public virtual A { // HERITAGE VIRTUEL
public:
  int b;
  B(int i, int j)
    :A(i), b(j)
    ...
  }
};

class C: public virtual A {
public:
  int c;
  C(int i, int j)
    :A(i), c(j)
  ....
  }
};
```

Classe A virtuelle (2)

Pour que ça fonctionne bien, il faut aussi faire un **appel explicite** au constructeur de `A` dans la classe la plus dérivée (`D`) :

```
class D: public B, public C {
public:
  int d;
  D(int i, int j, int k, int l)
     // APPEL AU CONSTRUCTEUR DE LA CLASSE VIRTUELLE
    :A(i),B(i,j), C(0,k), d(l)
  ...
  }
};
```

☞ les appels au constructeur de `A` dans les classes intermédiaires sont alors ignorés et l'exécution est la suivante...

Classe A virtuelle (3)

```
Création de A
Création de B
Création de C
Création de D
A: 1
C: 3
A: 1
B: 2
D: 4
Destruction de D
Destruction de C
Destruction de B
Destruction de A
```