

Programmation Orientée Objet

- Programme = système dynamique mêlant données structurées et opérations sur ces données
- Programmation = organisation des données (structures) et définition d'opérations (fonctions)
- La POO est un **modèle d'abstraction** des rouages d'un tel système :
 - pour faire apparaître plus clairement les tenants et les aboutissants
 - pour formaliser les méthodes de conception
 - pour masquer les détails inutiles

Programmation Orientée Objet

> Modélisation Objet

- **Modélisation** basée sur la notion d'**objet** qui combine de façon cohérente un ensemble de données et les opérations associées
- Comme toute entité physique, un objet a un **état** (structures des données) et un **comportement** (opérations)
- Avantages
 - modularité naturelle dès la conception du programme
 - abstraction et protection des structures de données (encapsulation)
 - meilleure lisibilité et réutilisation accrue des modules



Montre

état :

date, heure,
charge de la pile

comportement :

afficher l'heure
avancer l'heure
reculer l'heure

...



Bouteille

état :

ouvert, fermé
t° du contenu
vol. du contenu

comportement :

ouvrir / fermer
verser / remplir

...

$$a = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Vecteur

état :

dimension
liste des coefficients

comportement :

ajouter un vecteur
produit vectoriel
mult. / matrice

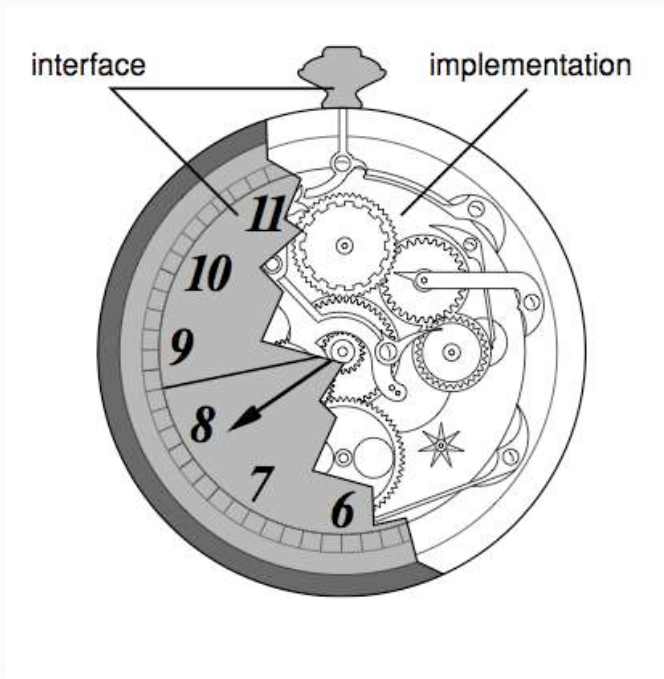
...



Modélisation Objet

> Interface & implémentation

- Comment formaliser cette approche par le langage ?
- **Interface** : point de vue de l'**utilisateur** sur l'objet
 - “ce que c’est” (sa fonction)
- **Implémentation** : point de vue du **concepteur-programmeur**
 - sa structure interne (“comment ça marche”)



Interface & implémentation

> Extension du modèle procédural

- **Couplage des structures de données et fonctions** utilisées en langage C.
 - **L'interface** de l'objet est l'ensemble des fonctions (en-têtes C) applicables à une structure
 - **L'implémentation** regroupe l'organisation des données (struct) et la définition des fonctions (en-têtes + code)
- La structure des données peut être masquée à l'utilisateur (elle devient un détail de l'implémentation)
 - on y accède indirectement via l'interface
 - l'utilisateur se concentre uniquement sur le comportement



Interface *Vecteur*

```
new(entier dim);  
delete();  
  
coeff(entier c) : réel;  
add(vecteur v);  
prod_vect(vecteur v);  
prod_scalaire(vecteur v) : réel;  
mult(réel r);  
...
```

Implémentation *Vecteur*

```
structure :  
    dim : entier  
    coeffs : tableau de réels  
  
fonctions :  
    mult(réel r) {  
        pour  $i$  de 1 à  $dim$  faire  
            coeffs[i] <- (coeffs[i] x r)  
    }
```

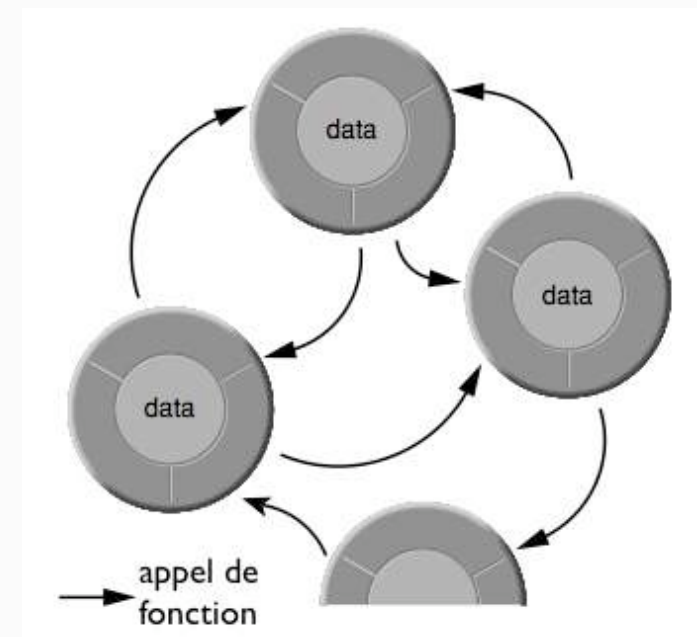
....



Modèle objet

> Structuration et concepts

- Les **attributs** (*data members*) sont les “champs” de données qui définissent l'**état d'un objet**
- Les **méthodes** (*methods*) sont les “fonctions membres” qui définissent son **comportement**
- Un programme est un réseau dynamique d'objets connectés



Modèle objet

> Classe

- Manipuler plusieurs objets de même nature (entiers, vect...)
- Extension de la notion de type en POO : **une classe est** la définition d'**un type d'objet**
- Une classe est une forme d'**espace de nom**
- Une **instance** est un objet qui existe en mémoire sur le modèle d'une classe donnée

Classe *Vector3* :

Vecteur : $\begin{bmatrix} x : \text{entier} \\ y : \text{entier} \\ z : \text{entier} \end{bmatrix}$

Instances de *Vecteur* :

$v1 = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}$ $v3 = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}$ $v2 = \begin{bmatrix} -1 \\ 2 \\ 9 \end{bmatrix}$



Classe > Instances

- **Différentes instances** d'une même classe **disposent de données spécifiques** structurées par le même modèle (la classe) mais contiennent des valeurs potentiellement différentes
- **Différentes instances** d'une même classe **partagent les mêmes méthodes** (code) : elles ont le même comportement
 - les vecteurs v1 et v2 ont des valeurs différentes, mais on peut leur appliquer la multiplication par un scalaire de la même façon
- Deux objets qui ont le même état mais ne partagent pas les mêmes méthodes ne peuvent pas être de la même classe

Modèle objet

> Modularité

- Modularité fonctionnelle
 - Langage C : un module = un fichier source (compilation séparée, variables *static*...)
 - C++ : un module = une classe (= en général un fichier source)
- «Réutilisabilité»
 - Partage d'interface entre différentes classes aux comportements similaires (ex: vecteur / matrice pour la mult. par un scalaire)
 - Solution intégrée à un problème (ex : opérations sur les vecteurs)
 - Indépendance données / interfaces (ex : plusieurs implémentations possibles pour une même interface)

Modèle objet

> Encapsulation

- Objectif : indépendance totale entre programmeur et utilisateur d'un objet
 - **structures internes complètement masquées à l'utilisateur**, accessible uniquement via l'interface (fixée au départ)
 - les optimisations, corrections de bogues et autres changements d'algorithmes n'affectent pas l'utilisateur
- Importance d'une phase de conception globale et réfléchie
 - attention au choix des classes / besoins
 - l'interface ne doit pas être modifiée (ou exceptionnellement)

46

Interface

$$a : \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

```
new(entier val=0);
delete();
```

```
coeff(entier i) : réel;
```

```
...
```

Implémentation n°1

$$a = (x, y, z)$$

attributs :

```
réel x, y, z;
```

méthodes :

```
coeff(entier i) {
    si (i == 1)
        retourner x;
    sinon si (i == 2)
        retourner y;
    sinon
        retourner z;
}
```

```
...
```

Implémentation n°2

$$a = [\square_0 \square_1 \square_2]$$

attributs :

```
réel a[3];
```

méthodes :

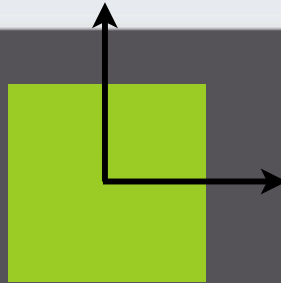
```
coeff(entier i) {
    retourner a[i-1];
}
...
```

Plusieurs implémentations possibles
pour une même interface

Modèle objet

> Polymorphisme

- Propriété qui permet à différentes classes de partager une interface commune
 - deux classes peuvent avoir une ou **plusieurs méthodes de même nom** (une classe est un “espace nommé”)
 - deux méthodes d’une même classe peuvent avoir le même nom et des paramètres différents
 - impossible en C : deux fonctions différentes = deux noms différents
- Simplification de l’interface
 - réutilisation de conventions (ex : multiplication par un vecteur/matrice)



Carré

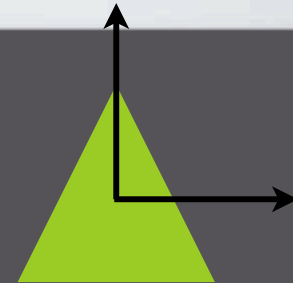
Interface

dessiner()
dessiner(couleur : entier)

Implémentation

x, y, L : réel

```
dessiner() {  
    allerÀ( $x-L/2, y-L/2$ )  
    tracerVers( $x+L/2, y-L/2$ )  
    tracerVers( $x+L/2, y+L/2$ )  
    tracerVers( $x-L/2, y+L/2$ )  
    tracerVers( $x-L/2, y-L/2$ )  
}
```



Triangle (équilatéral)

Interface

dessiner()
dessiner(couleur : entier)

Implémentation

x, y, L : réel

```
dessiner() {  
    allerÀ( $x-L/2, y-L\tan(\pi/6)/2$ )  
    tracerVers( $x+L/2, y-L\tan(\pi/6)/2$ )  
    tracerVers( $x, y+L/2\cos(\pi/6)$ )  
    tracerVers( $x-L/2, y-L\tan(\pi/6)/2$ )  
}
```



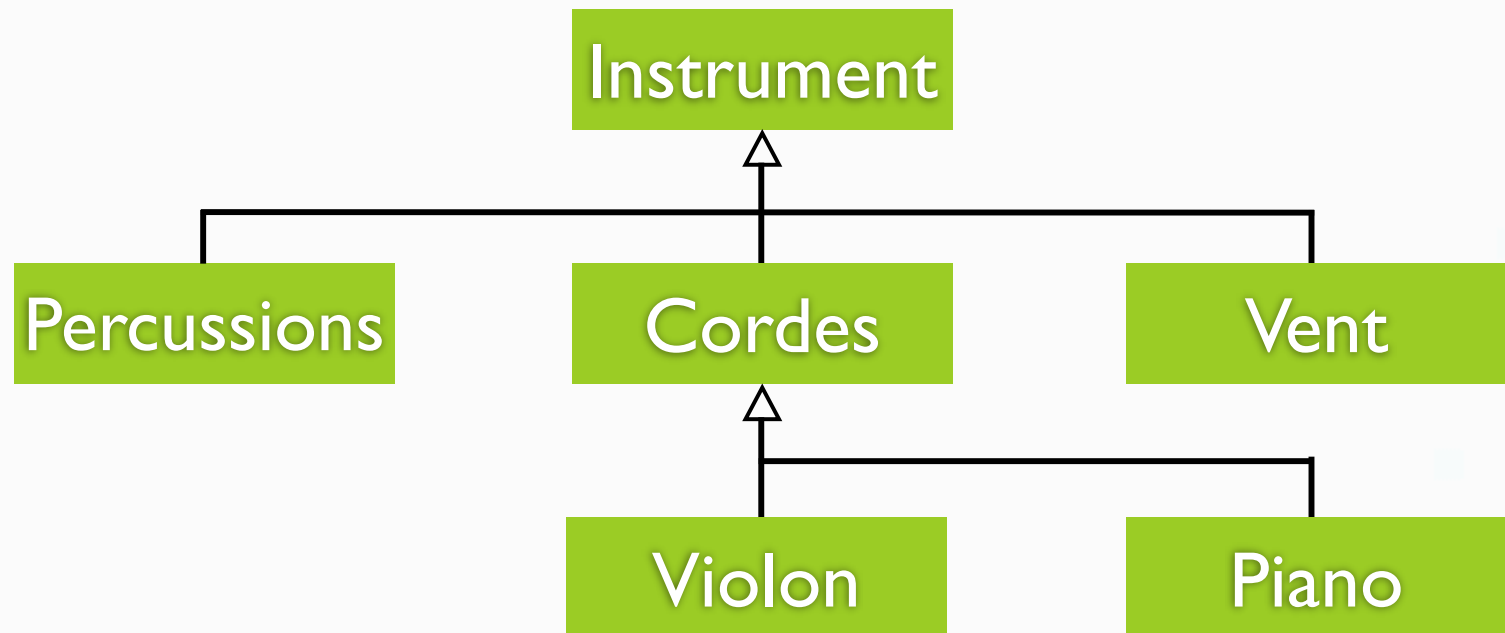
Modèle objet

> Héritage (spécialisation)

- ■ Propriété qui permet de construire de nouvelles classes d'objet à partir de classes existantes
 - ■ la classe existante est appelée **classe de base**
 - ■ la nouvelle classe est appelée **classe dérivée**, elle **hérite de la structure (attributs) et du comportement (méthodes)** de la classe de base
- ■ But : enrichir la classe de base
 - ■ en ajoutant de nouvelles méthodes ou de nouveaux attributs
 - ■ en remplaçant des méthodes existantes
 - ■ en étendant des méthodes existantes

Héritage

> Diagramme de classes



↑ = hérite

Instrument

Structure interne

string nom, marque;

Interface

init(string nom);

setMarque(string m);
jouerNote(entier n);

Instrument



Cordes

Structure interne

string nom, marque;
réel cordes[];

Interface

init(string nom);

setMarque(string m);
jouerNote(entier n);
setCordes(réel r[]);

Cordes



Piano

Structure interne

string nom, marque;
réel cordes[];
entier type;

Interface

init(string nom);

setMarque(string m);
jouerNote(entier n);
setCordes(réel r[]);
setType(entier t);

setMarque = hérité

jouerNote = redéfini

nouvRayon = nouveau

↑ = hérite (*inherits*)

Héritage

> Utilisations

- Réutilisation de code en “factorisant” le code commun dans une classe de base
- Mettre en place un “protocole” (interface à laquelle doivent se conformer des classes dérivées)
- Créer des modules de fonctionnalités génériques (à spécialiser pour ses propres besoins)
- Apporter de légères modifications (*customisation* d’UI)
- Expérimenter (tester de nouvelles implémentations)

Héritage & polymorphisme

> Liaison dynamique

- Une instance de classe dérivée est aussi une instance de la classe de base correspondante
 - on peut la manipuler comme telle
 - on peut affecter un objet dérivé à un objet de la classe de base
 - on peut appeler les méthodes de la classe de base
- Cas d'une méthode redéfinie d'un objet manipulé comme une instance de la classe de base
 - comportement par défaut : le code de la classe de base est appelé
 - méthodes virtuelles : le code redéfini (classe dérivée) est appelé

Langage C

```
faireJouer (void *inst, int type) {  
    switch (type) {  
        case VIOLON:  
            jouerViolon((Violon*)inst);  
            break;  
        case PIANO:  
            jouerPiano((Piano*)inst);  
            break;  
        etc...  
    }  
}  
...  
faireJouer ((void*)monPiano, PIANO);
```

en C, on devrait appeler une fonction adaptée pour chaque type d'objet passé en paramètre : **il faut donc un moyen de tester le type de l'objet**

Langage C++

```
faireJouer (Instrument& inst) {  
    inst.jouer();  
}  
...  
faireJouer ( monViolon );  
faireJouer ( monPiano );
```

code utilisé pour la méthode "faireJouer" :

- celui de *Instrument* ?
- celui de l'objet passé en paramètre (*Violon*, *Piano*) ?

Modèle objet

> Composition & Agrégation

Vecteur



Matrice

Chercheur



Département

- **Composition : relation d'appartenance** entre deux classes
 - ex : une matrice 4x4 est composé de 4 vecteurs de dimension 4
 - la création (*resp.* destruction) d'une instance de la classe qui possède (matrice) entraîne la création (*resp.* destruction) de la seconde (vecteur)
- **Agrégation : relation sans appartenance** entre deux classes
 - ex : un département scientifique est composé (entre autres) de chercheurs
 - un chercheur peut aussi travailler dans un autre département
 - la disparition du département n'entraîne pas celle de ses chercheurs !!

Templates (C++)

> Classes et méthodes génériques

- Objectifs : concevoir des classes ou des fonctions génériques applicables à différents types de base
 - on écrit une seule fois le code pour un type générique T
 - le code est instancié par le compilateur pour tous les types particuliers souhaités (et compatibles)
- Très utile pour les structures de données classiques (*cf.* STL)
 - tableaux dynamiques, listes, tables de hachage, arbres *etc...*
 - applicable pour des types simples (int, float...) mais aussi des classes complexes

Modélisation objet

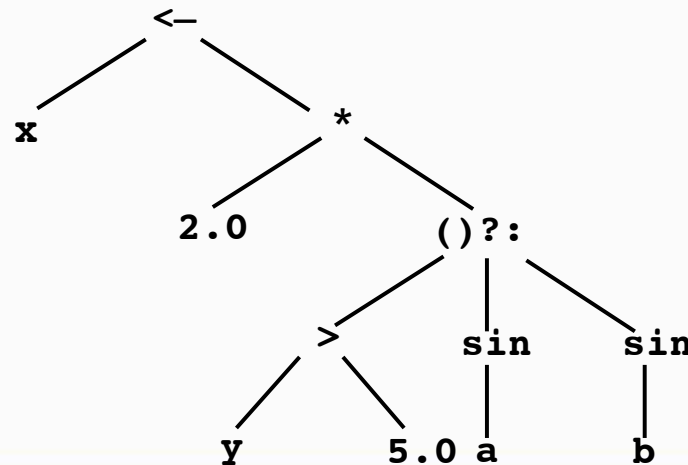
> Exemple

- Programmation d'un interpréteur d'expressions arithmétiques
 - objectif : créer un programme fonctionnel qui met en œuvre la (presque) totalité des concepts de POO !
 - mettre à disposition de l'utilisateur des outils de programmation simples pour la création, l'affichage et l'évaluation d'expressions évoluées
 - utilisation de **variables** : le seul type numérique utilisé est le réel (*double*)
 - expressions évoluées : les **affectations** dans des variables, les **expressions booléennes**, une **expression if-then-else** qui retourne le résultat de l'expression branchée, une **boucle for** !..

Expressions arithmétiques

> Représentation

- Une expression est représentée sous la forme d'un arbre dont chaque nœud est une sous-expression particulière
- Exemple : $x \leftarrow 2.0 * ((y > 5.0) ? \sin(a) : \sin(b))$





Modélisation objet de l'interpréteur

> Objets

Expression	Const	Variable
- une représentation textuelle	- une représentation textuelle - une valeur réelle	- une représentation textuelle - une valeur
- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> valeur de la var.) - affecter une valeur - obtenir représentation textuelle / afficher

Unary	Binary	Print
- une représentation textuelle - une sous-expression	- une représentation textuelle - deux sous-expressions	- une représentation textuelle - une sous-expression
- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher	- initialiser - évaluer - obtenir représentation textuelle / afficher

59



Modélisation objet de l'interpréteur

> Objets

60

Cos...	Addition...	Affectation
- une représentation textuelle - une sous-expression	- une représentation textuelle - une sous-expression	- une représentation textuelle - une variable - une sous-expression
- initialiser - évaluer (cosinus) - obtenir représentation textuelle / afficher	- initialiser - évaluer (somme) - obtenir représentation textuelle / afficher	- initialiser - évaluer (affecter la variable) - obtenir représentation textuelle / afficher

If-Then-Else	While	Block
- une représentation textuelle - une expression booléenne - deux sous-expressions	- une représentation textuelle - une sous-expression	- une représentation textuelle - une séquence de sous-expressions
- initialiser - évaluer (-> expr. branchée) - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> dernière éval) - obtenir représentation textuelle / afficher	- initialiser - évaluer (-> dernière éval) - obtenir représentation textuelle / afficher

Modélisation
objet de
l'interpréteur

> Diagramme
de classes

