27

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 tout 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
- ...

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

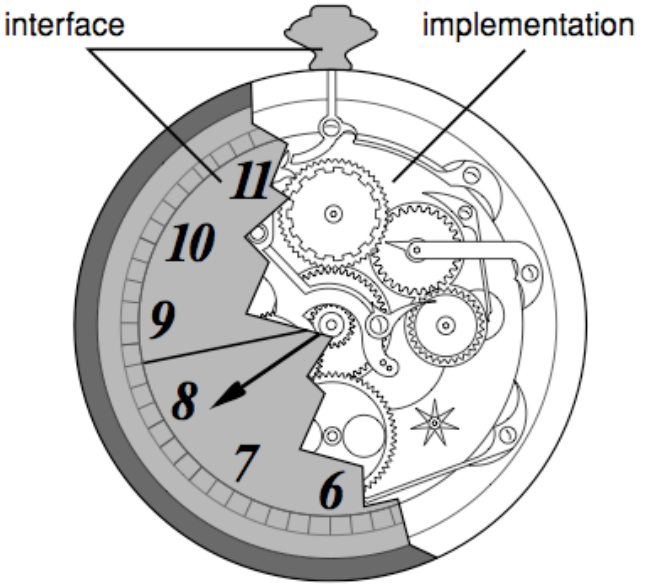
Vecteur

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

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

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);
...

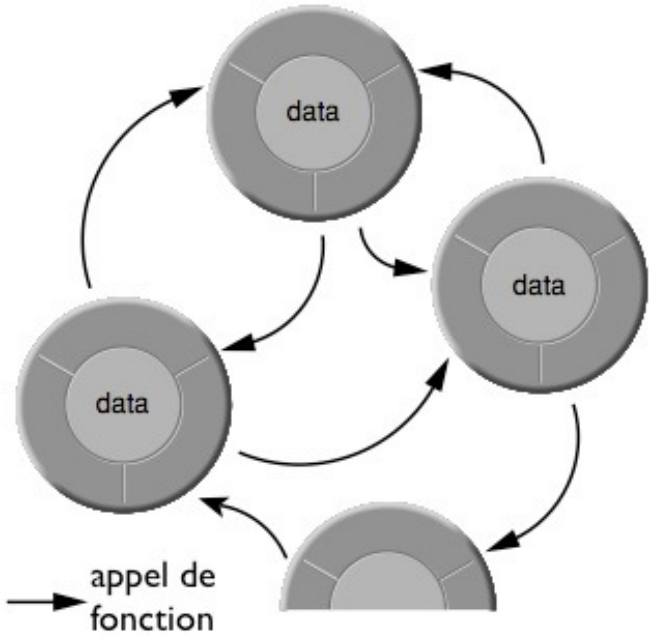
Implémentation *Vecteur*

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* :

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

Instances de *Vecteur* :

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

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 même 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)

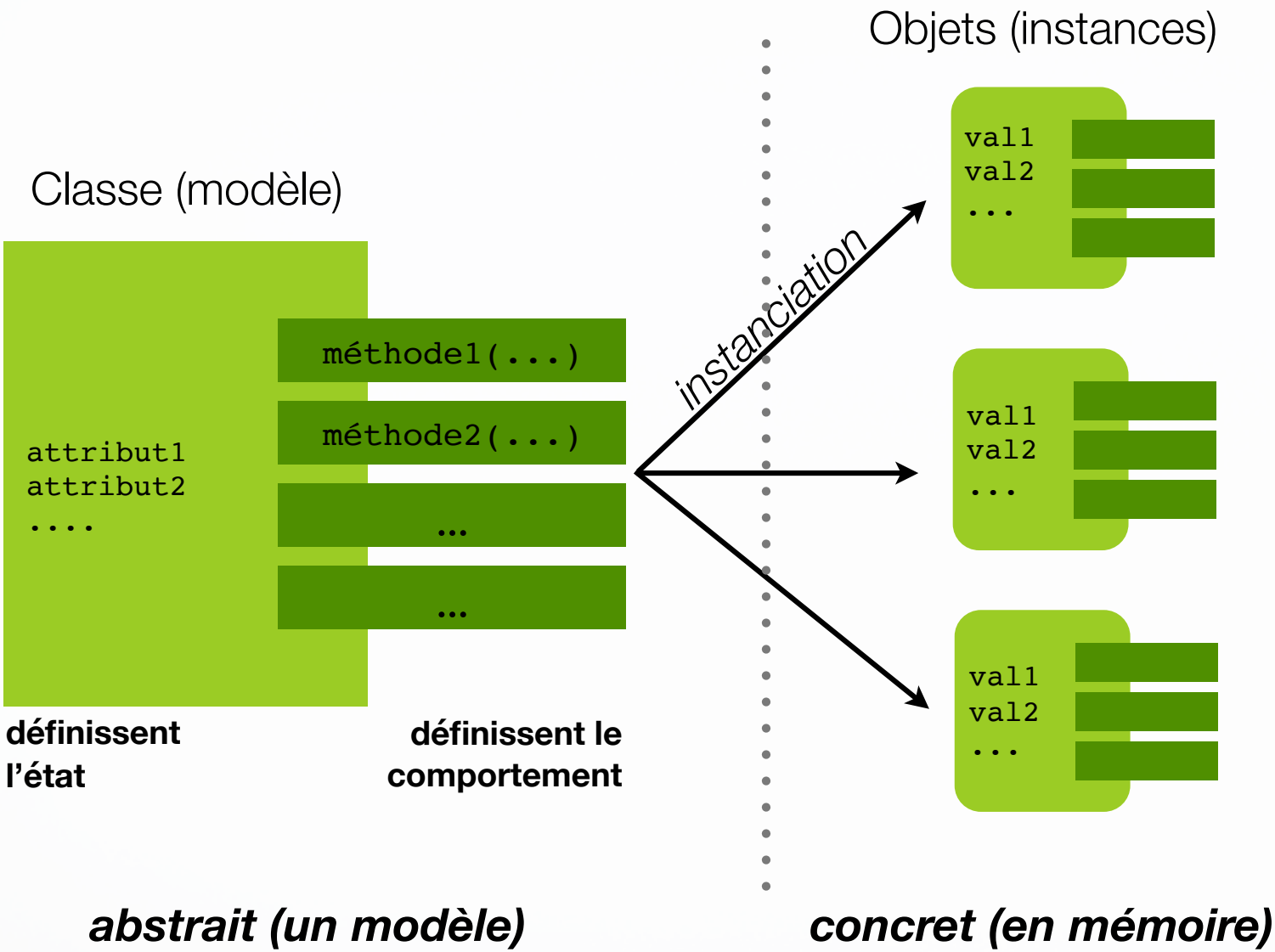
Modèle objet
> Encapsulation

- 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 changement 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)

Interface	Implémentation n°1	Implémentation n°2
$a : \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$	$a = (x, y, z)$	$a = [\square_0 \square_1 \square_2]$

new(entier val=0);
delete();
 	
coeff(entier i) : réel;
 	
...
	attributs :
	réel x, y, z;
	
	méthodes :
	coeff(entier i) {
	si (i == 1)
	retourner x;
	sinon si (i == 2)
	retourner y;
	sinon
	retourner z;
	}
	...
		attributs :
		réel a[3];
		
		méthodes :
		coeff(entier i) {
		retourner a[i-1];
		}
		...

Plusieurs implémentations possibles pour une même interface



Les classes C++

- Permet de définir une classe conforme au modèle structurel d'un objet
  - **attributs** (comparables à des champs de structure)
  - des **méthodes** (les "fonctions membres")
- L'encapsulation des données est assurée par des **droits d'accès aux attributs et aux méthodes**
  - **public** : accessible depuis partout
  - **protected** : pas d'accès "de l'extérieur" sauf depuis les classes dérivées
  - **private** (par défaut) : accessible qu'en interne

Les classes C++
> Déclaration / définition

- **Déclaration** : en général dans un fichier en-tête (`.h`) / contient **les attributs et des prototypes des méthodes**
- **Définition des méthodes** : dans le fichier source (`.cc`)
- Remarque : une méthode a directement accès à tous les attributs et toutes les autres méthodes quels que soient leur droit d'accès (`private`, `protected` ou `public`)
- On accède aux membres (f° ou attributs) à partir d'une instance avec la même syntaxe que l'accès aux champs d'une `struct`

Les classes C++
> Déclaration d'une classe

```
// fichier car.h
#include <string>

using namespace std;

class Car {
    // attributs (privés par défaut)
    string  _name;
public:

    // méthodes publiques
    string  getName() const;                // accesseur (getter)
    void    setName(const string& name);    // mutateur (setter)
    void    print() const;

}; // ne pas oublier le ';' à la fin
```

syntaxe de déclaration d'une nouvelle classe :
`class NomDeLaClasse { };`

droits d'accès (ici public) pour les attributs ou méthodes qui suivent

Instanciation automatique de la classe *Car*

L'inclusion des fichiers en-têtes et la fonction principale sont les mêmes qu'en langage C

```
// fichier main.cc

#include "car.h"

int main (int argc, char * const argv[]) {
    // insert code here...

    Car aCar;   // on crée une instance de la classe Car

    aCar._name = "Clio";     // ERROR : _name is private
    aCar.setName("Clio");   // on lui affecte un nom

    aCar.print();           // on affiche son nom

    return 0;
}
```

une méthode s'applique à une instance donnée via l'opérateur d'indirection **.**

en sortie

```
> Clio
```

Instanciation dynamique de la classe *Expression*

```
// fichier main.cc

#include "car.h"

int main (int argc, char * const argv[]) {
    // insert code here...

    Car *aCarPtr;       // pointeur vers un objet de type Car
    aCarPtr = new Car; // allocation dynamique

    aCarPtr->_name = "Clio";     // ERROR : name is private
    aCarPtr->setName("Twingo");  // on lui affecte un nom

    aCarPtr->print();                  // on affiche son nom

    delete aCarPtr;

    return 0;
}
```

une méthode s'applique au pointeur d'une instance donnée
via l'opérateur d'indirection **->**

Définition des méthodes de la classe *Car*


```
// fichier car.cc

#include "car.h"
#include <iostream>

string Car::getName() const {
    return _name;
}

void Car::setName(const string& name) {
    _name = name;
}

void Car::print() const {
    cout << getName() << endl;
}
```

les méthodes accèdent directement aux attributs de l'instance sur laquelle elles sont appliquées

une méthode peut appeler directement toute autre méthode de la même classe, qui s'applique alors implicitement à la même instance

Méthodes `inline`

Remplace une fonction courte par son code au niveau de l'appel (pour améliorer les performances)

```
// fichier car.h
class Car {
   // attributs (privés par défaut)
   ...

public: // méthodes publiques
   ...
  void print() const {
   cout << getName();
  }
};


inline void Car::print() const
{
   cout << getName();
}
```

définition de la méthode directement dans la déclaration de la classe...

ou...

à l'extérieur de la classe (toujours dans le fichier `.h`) en qualifiant la méthode avec le mot-clé `inline`

Les classes C++ > Méthodes `const`

- Les méthodes **`const`** ne modifient pas l'état de l'objet

```
// fichier car.h
class Car {
  string _name;

public:
  void print() const;
  ...
};


// fichier car.cc
void Car::print() const
{
  cout << getName();
}
```

la présence du mot-clé **`const`** à la fin de la signature d'une méthode (dans sa déclaration ET dans sa définition) **garantit que cette méthode ne modifiera pas l'état de l'objet**

Permet un **contrôle strict de l'accès aux données**

```
// *** cas (1) const ***
void Car::print() const {
   cout << getName();
   _name = "4L"; // ERROR : assignment of read-only location
}

void printCarName(const Car& c) {   // fonction externe
  c.print(); // OK ! car print() est const
}


// *** cas (2) pas const ***
void Car::print() {
   cout << getName();
   _name = "2CV"; // OK ! car print() n'est pas const
}

void printCarName(const Car& c) {   // fonction externe
  c.print();   // ERROR : passing 'const Car' as 'this' argument
               // of 'void Car::print()' discards qualifiers
}
```

- Il est possible de surcharger une méthode `const`

```
class Vector3 {
  double _coeff[3];

public:
  ...
  double& coeff(unsigned i) { return _coeff[i]; }       // (1)
  double coeff(unsigned i) const { return _coeff[i]; }  // (2)
};
```

les méthodes `(1)` et `(2)` sont considérées comme deux méthodes différentes

```
#include <iostream>
void initVector(Vector3& v) {
  for (int i=0; i < 3; i++)
    v.coeff(i) = 0.0; // appel de la méthode (1)
}

void printVector(const Vector3& v) {
  cout << "[" << v.coeff(0) << "," << v.coeff(1)
       << "," << v.coeff(2) << "]" << endl; // appels de (2)
}
```

ici **c'est la méthode `(1)` qui est appelée** car on modifie les attributs du vecteur `v`

dans ce cas **c'est la méthode `(2)` qui doit être appelée** car elle garantie que l'état de `v` ne sera pas modifié

Les classes C++
> Surcharge de méthode

```
// fichier car.h
class Car {
    // attributs
public:
    ...

    //attribue le nom à partir d'une chaine de caractères
    void setName(const string& name);

    //attribue me nom à partir d'une chaîne et d'un n° de version
    void setName(const string& name, const double version);

};
```

surcharge de la méthode
`setName`
avec d'autres arguments

Les classes C++
> Pointeur `this`

- Chaque fois qu'une fonction membre est appelée, **un pointeur** nommé **`this`** lui est implicitement et automatiquement **passé en paramètre**
- Il **contient l'adresse de l'objet courant** (sur lequel est appliqué la méthode) et il **ne peut pas être modifié**

```
void Car::setName(const string& name);
```

équivalent à

```
void Car_setName(Car *this,const string& name);
```

Pointeur `this` > Exemple (1)

```
#include "car.h"

string getName() {                  // fonction externe nommée getName()
   return "Un nom générique";
}


string Car::getName() const         // méthode de la classe Car
{
   return _name;
}


void Car::print() const
{
   cout << this->getName(); // <=> cout << getName();
}
```

l'utilisation de `this` permet de **lever toute ambiguïté éventuelle sur la fonction `getName()` : il s'agit d'une autre méthode du même objet de type Car**

s'il n'y a pas d'ambiguïté l'utilisation de `this` est facultative

Pointeur `this`
> Exemple (2)

```
#include <iostream>
using namespace std;

class Vector3 {
  double _coeff[3];

public:
  void saisie() {
    double _coeff[3];

    cin >> _coeff[0] >> " " >> _coeff[1] >> " " >> _coeff[2];
    for (int i=0; i < 3; i++)
        this->_coeff[i] = _coeff[i];
  }
};
```

une variable locale de méthode porte le même nom qu'un attribut (c'est possible)

l'utilisation de `this` permet d'accéder à l'attribut plutôt qu'à la variable locale

Pointeur `this`
> Exemple (3)

```
#include "vector3.h"

Vector3& Vector3::product(double s) {

  for (int i=0; i < 3; i++)
   _coeff[i] = (_coeff[i] * s);

  return *this;
}
```

l'utilisation de `this` permet de **retourner une référence sur l'objet concerné par l'application de la méthode `product`**

Les classes C++
> Constructeur

- **Méthode spécifique** de la classe qui sert à initialiser automatiquement et correctement un objet (instance) à sa création
- Un constructeur **porte le nom de la classe, ne retourne rien** (même pas `void`) et peut contenir n'importe quels paramètres
  - ex: `Car(const string& name)`
- Un constructeur qui n'a aucun paramètre ou uniquement des paramètres par défaut est dit **constructeur par défaut**
  - ex: `Car(const string& name = "")`
- Un constructeur par défaut (sans paramètre) existe - par défaut - si aucun autre constructeur n'est défini !
- **On peut surcharger un constructeur** (comme toute autre méthode)

Constructeurs

```
// fichier car.h
class Car {

    string  _name;
public:
    //constructeur
    Car(const string& name="Voiture");

    string  getName() const;
    void    setName(const string& name);
    void    print() const;
};
```

déclaration du constructeur `Car` qui est aussi constructeur par défaut (chaîne de caractère «Voiture»)

définition du constructeur `Car :` on initialise les attributs

```
// fichier car.cc
Car::Car(const string& name) {
    _name = name;
}
```

Constructeurs

instanciation avec constructeur : on crée et on initialise une instance directement à partir des valeurs souhaitées

```
// fichier main.cc

int main (int argc, char * const argv[]) {

    Car aCar("Twingo");   // nouvelle instance de classe Car

    aCar.setName("Twingo");         // INUTILE !
    aCar.print();                   // on affiche sa description

    Car *aCarPtr;
    aCarPtr = new Car; // on appelle le const. par défaut


    aCarPtr->print(); // affiche «Voiture»

    return 0;
}
```

inutile de lui affecter une description car cela a déjà été fait par le constructeur

Constructeurs
> Pointeurs et tableaux

```
Vector3 v1;  // appel au constructeur par défaut (s'il existe !)
Vector3 v2(2.3, 3.6, 1.7); // appel au constructeur à 3 paramètres
Vector3 v3 = Vector3(5.2); // appel au constructeur à 1 paramètre

Vector3 *ptr1, *ptr2;      // ok, mais aucun constructeur appelé !
ptr1 = new Vector3;        // appel au constructeur par défaut
ptr2 = new Vector3(1.2, 1.3, 1.7);  // appel constructeur à 3 paramètres

Vector3 *tab1 = new Vector3[5];   // !! chaque objet du tableau est initialisé
                  // par un appel au constructeur par défaut (s'il existe)

Vector3 tab2[3] = { Vector3(1.0), Vector3(2.0, 3.0, 4.0) };
                  // initialisation explicite des 2 premiers objets
                  // constructeur par défaut (s'il existe!) pour le troisième
```

Constructeurs
> Construction de copie

- Problème : **initialiser un objet `Car` avec le même état** (valeurs d'attributs) **qu'une autre instance** de même classe
  - ex: `Car aCar = anOtherCar;`
- Un constructeur particulier est appelé : le constructeur de copie
  - syntaxe : `Car(const Car& aCar);`
- Appelé chaque fois qu'une copie d'objet est effectuée
  - lors d'un passage de paramètre par valeur à une f° (c'est une copie !)
  - quand la valeur de retour d'une fonction est un objet
- **Constructeur de copie par défaut** : copie membre à membre

Constructeur de copie
> Exemple (Car)

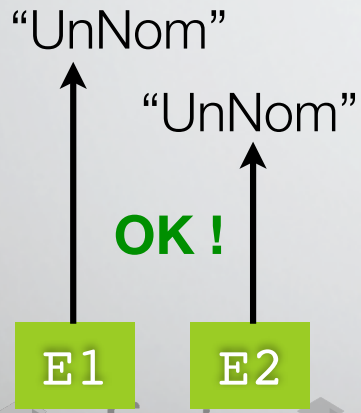
```
// fichier car.h
class Car {

    string  _name;
public:
    //méthodes publiques
    Car(const string& name="Voiture");

    Car(const Car& aCar) {
        _name = aCar._name;
    }

    ...

};
```

constructeur à un argument et par défaut : j'initialise l'expression à partir du paramètre

constructeur de copie (`inline`) : j'initialise la voiture comme une copie de la voiture `aCar` passée en paramètre

constructeur de copie facultatif ici car le constructeur par défaut a le même comportement

Construction de copie > Mémoire dynamique



- Le constructeur de copie est particulièrement utile pour la **copie des attributs alloués dynamiquement :**

```
class Car {    // car.h
   char *_name;
public:
   Car(const char* name) { _name = strdup(name); }
   Car(const Car& aCar);

   const char* getName() const { return _name; }
};

inline Car::Car(const Car& aCar) {
   _name = aCar._name;          // surtout pas !
   _name = strdup(aCar._name); // OK!
}
```

allocation et recopie de chaîne pour éviter que les attributs des deux objets pointent vers le même tableau de caractères en mémoire

Constructeurs
>Initialiser à la construction

- Exemple de la classe `Car`

les attributs sont initialisés à la file dans une liste séparée par des virgules
avant même l'accolade de début du code

```
class Car {
   double      _revision;    // n° de revision
   string      _name;        // un nom...
public:
   Car(double revision, string name="Voiture");
   ...
};

Car::Car(double revision, string name="Voiture") : _revision(revision),
_name(name)
{
   // code d'initialisation complémentaire
   _name = name;     // INUTILE (REDONDANT) !
}
```

comme `string` est une classe, alors `_name(name)` fera appel à un constructeur de copie de cette classe

Les classes C++
> Destructeur

- **Méthode spécifique** de la classe qui sert à “détruire” (libérer la mémoire...) correctement un objet à la fin de sa vie
- Un destructeur **porte le nom de la classe précédé de ~, ne retourne rien** (même pas `void`) et n’a aucun paramètre
  - ex: `~Car()` (on ne peut donc pas surcharger un destructeur)
- Les destructeurs ne sont pas appelés explicitement
  - automatiquement à l’utilisation de `delete` ou en fin de bloc de portée
- Un destructeur par défaut existe est généré par le compilateur
  - probablement pas bon en cas d’attributs alloués dynamiquement !
- **Rq** : constructeurs et destructeurs sont les seules méthodes non `const` à pouvoir être appelées pour des objets `const`

Destructeurs

```
class Car {
    string  _name;
public:
    Car(const string& name="Voiture");
    ...

    ~Car() { }
    ...
};
```

il n'y a rien de particulier à faire ici : la mémoire occupée par le chaîne `_name` sera libérée automatiquement après que son destructeur soit appelé

```
class Car {
    char  *_name;
public:
    Car(const char* name="Voiture");

    ~Car() { if (_name != NULL) free(_name); }
    ...
};
```

il faut libérer la mémoire allouée dynamiquement pour `*_name` car seul le pointeur `_name` lui-même sera libéré automatiquement

Constructeurs / Destructeurs
> Exemples (1)

```
// fichier car.h
class Car {
private:
   char *_name;
public:
   Car(const char* name);
   Car(const Car& c);
   ~Car();
   ...
};
```

```
// fichier car.cc
Car::Car(const char* name) {
   _name = strdup(name);
   cout << "construction de "<< _name << endl;
}

Car:: Car(const Car& c) {
   _name = strdup(c._name);
   cout << "copie de "<< _name << endl;
}

Car::~Car() {
   if (_name != NULL) {
      cout << "destruction de "<< _name << endl;
      free(_name);
   }
}
```

Constructeurs / Destructeurs
> Exemples (2)

```
// fichier main.cc

void fonction(Car c) {
  cout << "debut fonction" << endl;
  Car    c2("Deuxieme");
  cout << "fin fonction" << endl;
}

int main() {
  Car    c1("Premier");

  fonction(c1);
  cout << "fin main" << endl;
}
```

en sortie

```
> construction de Premier
> copie de Premier
> debut fonction
> construction de Deuxieme
> fin fonction
> destruction de Deuxieme
> destruction de Premier
> fin main
> destruction de Premier
```

ici **c'est la copie c de l'objet "Premier"**, passée en paramètre de la fonction, **qui est détruite**

Constructeurs / Destructeurs
> Visibilité

- **Constructeurs et destruteurs peuvent être `public` ou `private` (ou `protected`)**
  - permet d'interdire toute création explicite d'un objet (on doit passer par une fonction spéciale - par exemple méthode de classe)
  - de même un destructeur caché permet d'interdire explicitement des objets (avec `delete`)
- À manier avec précaution ! ;-)

Membres de classe
> Attributs `static`

- Comme un objet (instance) peut disposer d'attributs, une classe elle-même peut disposer d'**attributs de classe**
- **Ces attributs** sont instanciés une seule fois (au début du programme) et **sont partagés par toutes les instances**
  - sortes de "variables globales de classe"
  - ils sont affectés par les droits d'accès (`public`,`protected`,`private`)
  - ils sont accessibles (en lecture ou écriture) par toutes les instances de la classe et par l'extérieur s'ils sont `public`
  - *ex* : améliorer la lisibilité du programme, contrôler la création/destruction d'instances, compter le nombre d'instances, liste de toutes les instances...
- Ils sont qualifiés par le mot-clé `static`

Attributs `static` > Exemple (`Car`)

- Créer un compteur d'instances

```
// fichier car.h
class Car {
   // attributs et méthodes d'instance
   ...
public:
    static unsigned int _count;   // nombre d'instances de la classe Car
};

// fichier car.cc
// définition/initialisation d'un att. de classe
unsigned int   Car::_count=0;

// fichier main.cc
int main() {
   Car  myCar;
   ...
   cout << Car::_count << " instances de la classe Car" << endl;
}
```

on déclare un attribut de classe `public` de type `int` en le précédant du qualificatif `static`

on initialise un attribut de classe en dehors de sa déclaration (ici par exemple dans le fichier expression.cc)

on peut utiliser cet entier directement (préfixé par le nom de la classe) car il a été déclaré `public`

Attributs `static` > Exemple (`Car`)

- Créer un compteur d'instances

```
// fichier car.cc
Car::Car(const string& name) {

    _name = name;

    Car::_count ++; // on incrémente le compteur _count

}

Car::~Car() {

    Car::_count --; // on décrémente le compteur _count

}
```

une nouvelle instance : on incrémente le compteur d'instances

une instance va disparaître : on décrémente le compteur d'instances

Membres de classe > Méthodes `static`

- De même que des attributs de classe, une classe peut disposer de **méthodes de classe**
- Ces méthodes **ne peuvent accéder qu'aux attributs de classe**
  - impossible de faire référence à un attribut d'instance directement
- Elles **ne peuvent pas être surchargées**
- Elles **existent toujours** même s'il n'y a aucune instance de la classe
- Elles sont également **affectées par les droits d'accès** (`public`, `protected` et `private`)
- Elles sont également qualifiés par le mot-clé `static`

Méthodes `static` > Exemple (`Car`)

- On crée une méthode de classe pour consulter le compteur d'instances

```
// fichier expression.h

class Car {
// attributs et méthodes d'instance
public:
    ...

//attributs et méthodes de classe
protected:
    static unsigned long   _count;

public:
    static unsigned long getInstanceCount() { return _count; }
};
```

on déclare/définit une méthode de classe `inline` qui retourne le nombre d'instances de la classe `Expression`

l'attribut `_count` est maintenant protégé : il n'y a plus le risque de le modifier en dehors de la classe `Expression`

Méthodes `static` > Exemple (`Car`)

- On crée une méthode de classe pour consulter le compteur d'instances

```
// fichier car.h
class Car {
    ...
//attributs et méthodes de classe
protected:
    static unsigned long   _count;

public:
    static unsigned long getInstanceCount() { return _count; }
};

// fichier main.cc
int main() {
   Car  myCar;
   ...
   cout << Car::getInstanceCount() << " instances." << endl;
}
```

l'appel d'une méthode de classe n'est pas associé à une instance mais à une classe : **on préfixe par le nom de cette classe**

on ne peut plus utiliser directement l'attribut `_count` (protégé) : on appelle la méthode de classe qui retourne sa valeur

Membres de classe
> Exemple avancé (`Resource`)

- Créer un gestionnaire de ressources rudimentaire
  - compter et enregistrer les adresses de toutes les instances “en vie” dans un *pool*
  - retrouver l’adresse d’une instance par son nom
  - libérer toutes les instances qui ne l’ont pas été à la fin du programme
- Attributs de classe
  - *pool* (tableau) et décompte (entier) des pointeurs d’instances actives
- Méthodes de classe
  - initialisation et libération du *pool* (avec taille max statique)
  - ajout et retrait d’une instance dans le *pool*
  - recherche d’une instance par son nom

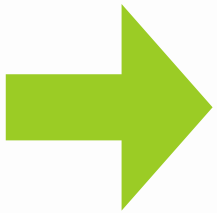
```
// fichier resource.h

#define POOL_SIZE   1024

class Resource {
  string   _name; // nom de l'instance
public:
   Resource(const string& name);
   Resource(const Ressource& r);
   ~Resource();

protected:
  static unsigned   r_count; // nombre d'instances !
  static Resource*  r_pool[POOL_SIZE]; // tableau de ptr de ressources (idem)

  static void count() { return r_count; } // retourne le nb d'instances de Resource
  static void initResourcePool(); // initialise le pool
  static void releaseAllResources(); // libère toutes les ressources
  static Resource* getResourceNamed(const string& name);

  static void registerResource(Resource *r); // enregistre r dans le pool
  static void unregisterResource(Resource *r); // retire r du pool
};
```

```
// fichier resource.cc

unsigned   Resource::r_count = 0;
Resource*  Resource::r_pool[POOL_SIZE];

void Resource::initResourcePool() {
  r_count = 0;
  for (unsigned i=0; i < POOL_SIZE; i++)
    r_pool[i] = NULL; // on initialise le pool (vide)
}

void Resource::releaseAllResources() {
  unsigned  i=0;
  // on supprime toutes les ressources restant dans le pool
  while (i < POOL_SIZE && r_count > 0) {
    if (r_pool[i] != NULL) {
        delete r_pool[i];
    }
    i++;
  }
}
```

```
// fichier resource.cc (suite)

void Resource::registerResource(Resource *r) {
  unsigned   count = 0, i=0;
  while (i < POOL_SIZE && r_pool[i] != NULL)
       i++; // recherche du premier emplacement vide disponible
  if (i < POOL_SIZE) {
       r_pool[i] = r;  // enregistrement de r
       r_count++;      // une ressource en plus dans le pool
  } else {
    //erreur : dépassement de capacité
   }
}

void Resource::unregisterResource(Resource *r) {
  unsigned   count = 0, i=0;
  while (i < POOL_SIZE && r_pool[i] != r)
       i++; // recherche de l'emplacement de r dans le pool
  if (i < POOL_SIZE) {
       r_pool[i] = NULL;  // on libère l'emplacement
       r_count--;         // une ressource en moins dans le pool
  }
}
```

```
// fichier resource.cc (fin)

Resource* Resource::getResourceNamed(const string& name)
{
  while ( (i < POOL_SIZE) && (r_pool[i]->_name != name) )
     i++; // recherche d'une ressource de nom "name"
  return (i < POOL_SIZE)? r_pool[i] : NULL;
}

Resource::Resource(const string& name) { // constructeur
  _name = name;
  registerResource(this);  // auto-enregistrement à la création
  cout << "Resource " << _name << " created and registered." << endl;
}

Resource::~Resource() { // destructeur
   unregisterResource(this); // auto-désenregistrement à la destruction
   cout << "Resource " << _name << " released and unregistered." << endl;
}
```

```
// fichier main.cc

int main() {
  Resource::initResourcePool();

  Resource *r = new Resource("-1-");
  r = new Resource("-2-"); //on perd l'adresse de -1- !
  r = new Resource("-3-"); //on perd l'adresse de -2- !!
   cout << "r est " << r->getName() << endl;

  //oups, j'ai besoin de retrouver l'adresse de la ressource -1- !
  r = Resource::getResourceNamed("-1-");
  cout << "et maintenant r est bien " << r->getName() << endl;
  cout << Resource::count() << "ressources ont ete creees." << endl;
  Resource::releaseAllResources();
}
```



```
> Resource -1- created and registered.
> Resource -2- created and registered.
> Resource -3- created and registered.
> r est -3-
> et maintenant r est bien -1-
> 3 ressources ont ete creees.
> Resource -1- released and unregistered.
> Resource -2- released and unregistered.
> Resource -3- released and unregistered.
```

Les classes C++

> Fonctions, méthodes et classes amies

- Objectif : permettre l'accès aux membres cachés d'une classe à une fonction ou méthode non membre, voire à une autre classe
  - pratique pour assouplir — à titre exceptionnel — l'encapsulation
  - *ex* : optimiser les méthodes très souvent appelées

```
void Matrix3::multiply(const Vector3& v) {
   for (int i=0; i < 3; i++)
     for (int j=0; j < 3; j++) {
        _coeff[j] = 0.0;
        for (int k=0; k < 4; k++)
          _line[i]._coeff[j] += m._line[i]._coeff[k] * tmp._line[k]._coeff[j]);
     }
}
```

on accède directement aux attributs des vecteurs pour éviter les appels de fonctions et copies inutiles

Méthodes/fonctions amies > “Déclaration d’amitié”

- On déclare le prototype de la fonction ou de la méthode “amie”, précédé du mot-clé **`friend`**, dans la définition de la classe

```
#include "factory.h"
class Car {
  unsigned _colorCode; // PRIVÉ !
public:
   ...

   // méthodes et fonctions amies
   friend void Factory::paintCar(const Car& c, unsigned color);
   friend void paintCar(Car& c, unsigned color);
};


void paintCar(Car& c, unsigned color) {
    c._colorCode = color; // accès à un attribut privé ok !
}
```

la méthode `paintCar` de `Factory` a maintenant accès aux attributs cachés de `Car`

la fonction `paintCar` a maintenant accès direct aux attributs cachés de `Car`

Méthodes/fonctions amies > “Déclaration d’amitié”

- Pour que toutes les méthodes d’une classe aient accès aux attributs cachés d’une autre, on peut la déclarer amie
  - *ex* : la classe **`Ecran`** pourrait être amie de la classe **`Expression`**

la classe `Factory` est amie de la classe `Car` : toutes les méthodes de `Factory` peuvent accéder aux membres cachés de `Car`

```
#include "factory.h"
class Car {
  unsigned _colorCode; // private
public:
   ...
   // classe(s) amie(s)
   friend class Factory;
   // méthode(s) amie(s)
   friend void paintCar(const Car& c, unsigned color);
};
```

  - **Attention, ceci constitue une brèche dans l’encapsulation**