

Programmation en C++

EFREI - 2016/2017

Nicolas Sicard

v4.1

2

Planning - Évaluations

- C++ (1) - Efrei / Esigetel - TI133P/UE51P - 4 ECTS
 - Cours : 6 séances (10h30)
 - TP : 8 séances (28h)
 - Évaluations : 1 DE (50%) - 1 TP (50%)
- C++ (2) - Efrei - TI152P/UE53LP3EF - 3 ECTS
 - Cours : 5 séances (8h45)
 - TP : 10 séances (31h30)
 - Évaluations : 1 DE (50%) - 1 TP (20%) - 1 PRJ (30%)

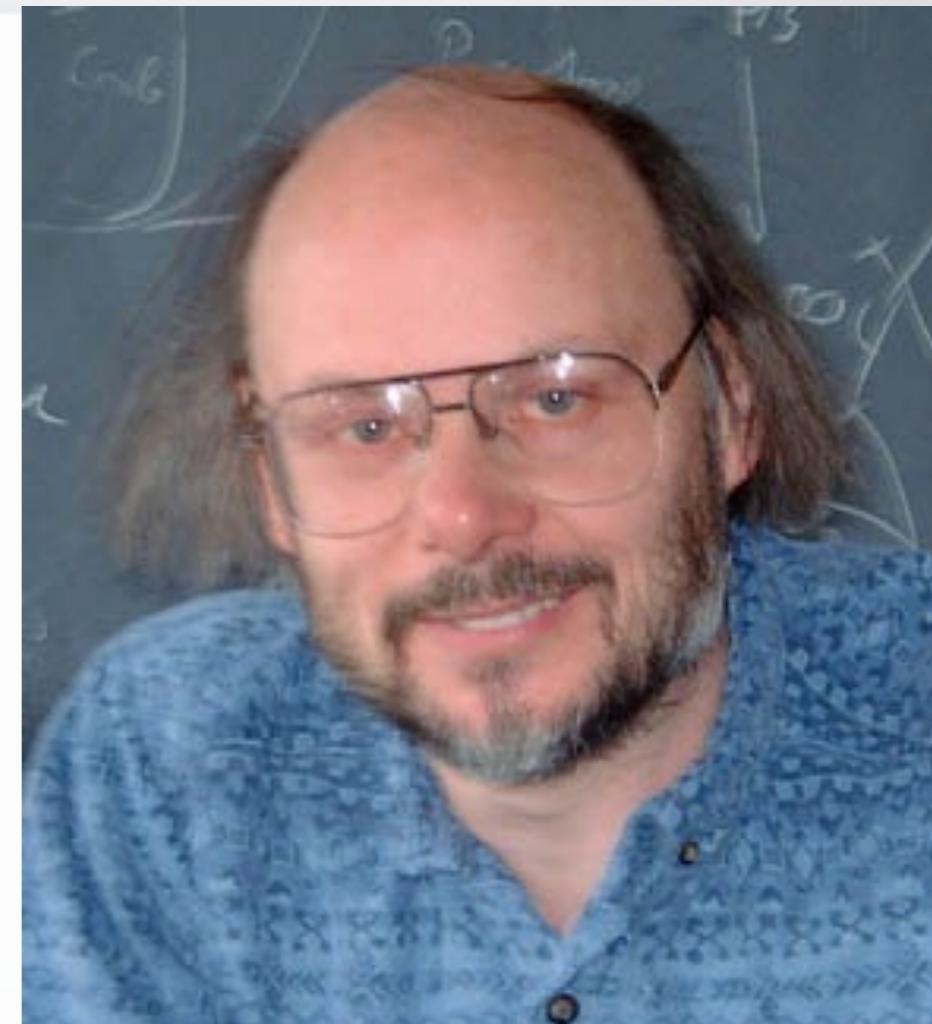
Références

- Thinking in C++ (2nd ed.) - *Bruce Eckel* - Prentice Hall
- Principes et pratique avec C++ - *Bjarne Stroustrup, Loïc Joly, Michel Michaud* - Pearson Education
- Beyond C++ Standard Library : An Introduction to Boost - *Björn Karlsson* - Addison Wesley

Généralités

> Historique

- 1980 - Bjarne Stroustrup (Bell Labs AT&T)
- Initialement “ C with classes ”
- Norme ANSI/ISO C++ en juillet 1998 (ISO/IEC 14882)
- Norme **C++11** publiée le 10 octobre 2011 !
- Combiner un langage classique populaire (C) avec un modèle de programmation moderne
- Compromis entre performance, réutilisation et abstraction



Enjeux

Oct 2016	Oct 2015	Change	Programming Language	Ratings	Change
1	1		Java	18.799%	-0.74%
2	2		C	9.835%	-6.35%
3	3		C++	5.797%	+0.05%
4	4		C#	4.367%	-0.46%
5	5		Python	3.775%	-0.74%
6	8	▲	JavaScript	2.751%	+0.46%
7	6	▼	PHP	2.741%	+0.18%
8	7	▼	Visual Basic .NET	2.660%	+0.20%
9	9		Perl	2.495%	+0.25%
10	14	▲	Objective-C	2.263%	+0.84%
11	12	▲	Assembly language	2.232%	+0.66%
12	15	▲	Swift	2.004%	+0.73%
13	10	▼	Ruby	2.001%	+0.18%
14	13	▼	Visual Basic	1.987%	+0.47%
15	11	▼	Delphi/Object Pascal	1.875%	+0.24%

5

Enjeux (2)

- Langage répandu : référence dans l'entreprise
- Pont entre le C et la programmation orientée objet (POO)
 - Concepts communs avec Java, C#, Objective-C...
- Langage portable : norme mise à jour régulièrement, compilateurs présents sur toutes les plateformes
- **Langage compilé : exécution rapide**

7

Compétences cibles

- Première partie
 - Modéliser un problème sous forme de classes
 - Programmer des classes en langage C++
 - Établir des relations (composition, agrégation, héritage) entre les classes pour mettre en œuvre un programme de bout en bout
 - Manipuler des fichiers en utilisant les flux C++
- Deuxième partie
 - Utiliser le polymorphisme et/ou les classes *templates* pour programmer des classes ou des fonctions génériques
 - Gérer les erreurs à l'aide des mécanismes d'exceptions
 - Connaître les principales nouveautés de la norme C++11



- Généralités
- Programmation orientée objet / Classes
- Surcharges d'opérateurs
- Composition, Agrégation, Héritage
- Flux et fichiers
- Polymorphisme et méthodes virtuelles
- Les exceptions
- Les “*templates*” et la *STL Library*
- C++11

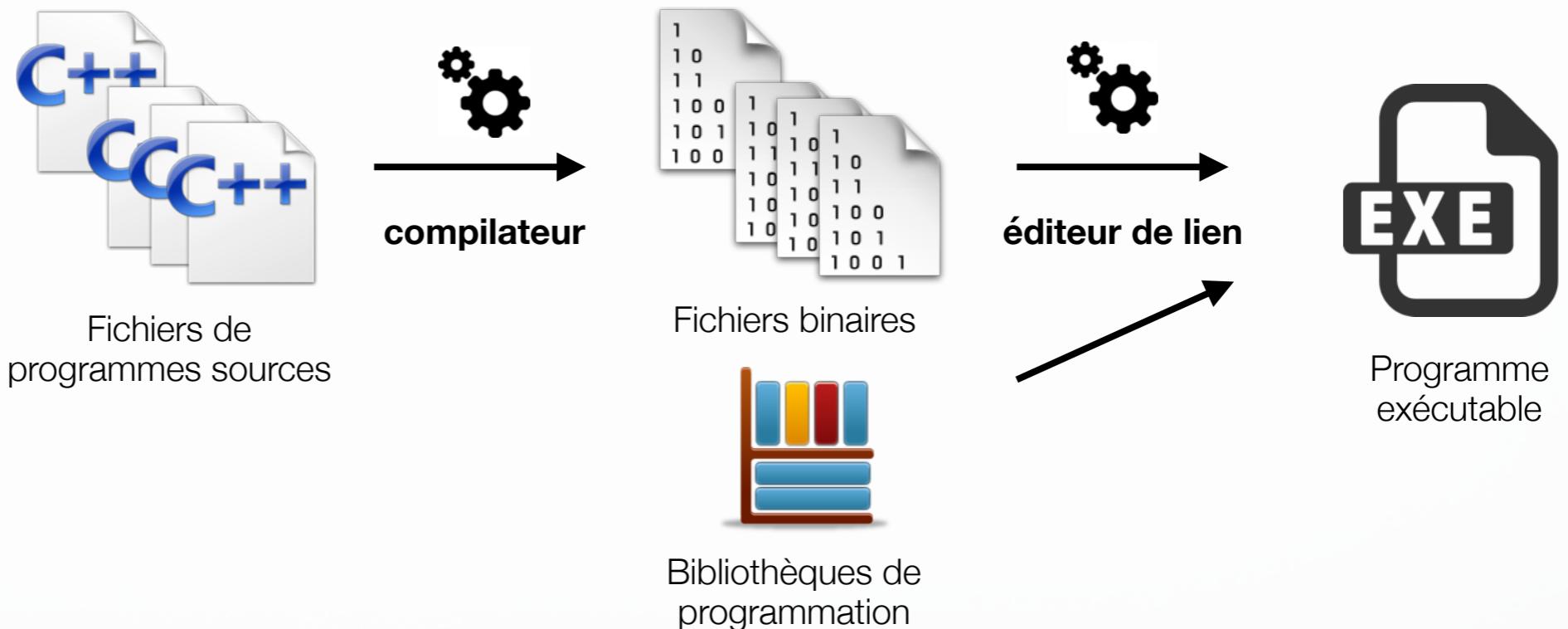
Première partie

Deuxième partie

Généralités

> Langage C / C++

- Un programme C ou C++ est compilé pour chaque plate-forme cible (\neq Java)



Généralités

> Langage C / C++

- C++ apporte certaines spécificités non liées à la POO
 - commentaires C : /* bloc de texte sur plusieurs lignes */
 - en C++ : //commentaire sur une seule ligne
 - transtypages de pointeurs doivent être explicites :

<pre>void* gen; int* adi;</pre>	<pre>/* en C */ gen = adi; adi = gen;</pre>	<pre>// en C++ gen = adi; adi = (int*)gen;</pre>
--	---	--

Généralités

> Langage C / C++

- N'importe quel fichier (.c ou .h) ou code source en langage C (propre) peut être compilé par un compilateur C++
- Les fichiers en-têtes portent également le suffixe .h et les fichiers sources l'en-tête .cpp ou .cc
- En ligne de commande (**UNIX**), le compilateur est **GNU g++** qui accepte presque toutes les mêmes options que **gcc**

Spécificités du C++

> Constantes (**const**)

- **En C et C++** : directive préprocesseur (avant compilation) pour définir des constantes - ou *macro* - en remplacement de texte
 - ex : `#define MAX 50`
- **En C++** : on utilise plutôt le mot-clé **const** pour définir une variable constante (non modifiable) correctement typée
 - exemple : `const int MAX=50;`
 - alors : `int tab[MAX]; // autorisé en C++ (pas en C) !!`
- Portées comparables à l'équivalent variable (sans **const**)

{

...
`int i = 0;`



}



En C++ on peut déclarer les variables n'importe où dans le code !

- La portée de la variable va de la ligne de sa déclaration jusqu'à la fin du bloc courant (y compris une boucle `for`)
- Une variable `const` peut être initialisée à sa déclaration à partir du résultat d'un calcul (mais n'est plus modifiable après)
- Intérêt : déclarer une variable au plus près de son utilisation offre une meilleure lisibilité



Spécificités du C++

> Références (synonymes)

- Permet d'attribuer un synonyme (alias)
 - à une variable : modifier le contenu d'une variable via un autre nom
 - un paramètre : remplacer le passage de paramètre par adresse
 - à une valeur de retour d'une fonction (au lieu d'un pointeur)
- Syntaxe
 - `int a;` // déclaration d'un entier
 - `int& b=a;` // `b` est synonyme de `a` (doit être initialisé à la déclaration)
 - `!! const int a;` puis `int& b=a;` n'est pas possible
 - !! ne pas confondre avec l'opérateur & (adresse de)

Références comme variables

```
double a = 1.0;
```

```
double& b = a;           // a et b désignent la même variable en  
mémoire !
```

```
double& c;              // ERROR : 'c' declared as reference but not  
initialized
```

```
a = 5.4;
```

```
cout << "a = " << a << " et b = " << b << endl;  
// affiche : a = 5.4 et b = 5.4
```

```
double *ptr = &b; //ptr <- adresse de b
```

```
*ptr = -2.3;
```

```
cout << "a = " << a << " et b = " << b << endl;  
// affiche : a = -2.3 et b = -2.3
```

- Références comme paramètres (remplace le passage par adresse)

```
//échange local seulement
void echangeCpy(int a, int b) { int tmp=a; a=b; b=tmp; }

//passage par adresse
void echangePtr(int *a, int *b) { int tmp=*a; *a=*b; *b=tmp; }

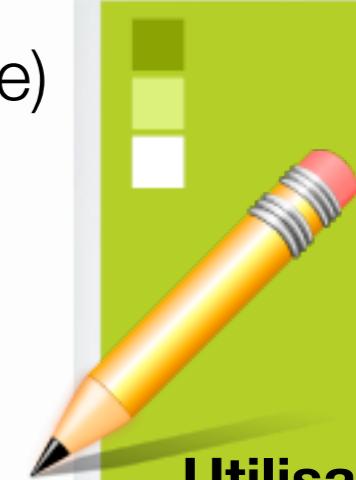
//passage par référence
void echangeRef(int& p, int& q) { int tmp=p; p=q; q=tmp; }

int main(){
    int a=3, b=5;

    echangeCpy(a,b); cout<<"a = "<<a<<" b = "<<b<<endl;
    // affiche : a=3 b=5

    echangePtr(&a,&b); cout<<"a = "<<a<<" b = "<<b<<endl;
    // affiche : a=5 b=3

    echangeRef(a,b); cout<<"a = "<<a<<" b = "<<b<<endl;
    // affiche : a=3 b=5
}
```



Utilisation des références en C++

16

Références

> Paramètres référence const

- On passe souvent une référence `const` en paramètre plutôt qu'une adresse :
 - passage par copie parfois problématique (données de grande taille)
 - passage par adresse parfois dangereux (modification de la mémoire) et syntaxe lourde (* et ->)
- Exemple : `typedef struct { double x; double y; } Point;`

on peut ainsi exercer un **contrôle strict de la modification des paramètres**

```
void affiche(const Point& p) {
    cout << "(" << p.x << "," << p.y << ")";
    p.x = 0.0; // ERROR : assignment of data-member
               'point::x' in read-only structure
}
```



Fonctions

> Arguments à valeur par défaut

- Attribuer une valeur par défaut à un ou plusieurs paramètres qui deviennent implicites à l'appel de la fonction
 - permet d'alléger l'écriture pour des valeurs souvent utilisées

```
void afficher(const Point& p, int couleur=0, const char *nom="Point") {  
    cout << nom << " = (" << p.x << "," << p.y << ")" << couleur << "]" << endl;  
}
```

attention, les paramètres par défaut doivent être en fin de liste !

```
int main() {  
    Point p1 = {0.0,0.0};  
    afficher(p1);  
    afficher(p1, 127);  
    affiche(p, "Zero"); // ERROR: invalid conversion from 'const char*' to 'int'  
    affiche(p1, 255, "Origine");  
  
    return 0;  
}
```



```
$ Point = (0.000000,0.000000)[0]  
$ Point = (0.000000,0.000000)[127]  
$ Origine = (0.000000,0.000000)[255]
```

Fonctions

> Surcharges de fonctions

- Une fonction peut être surchargée : **plusieurs fonctions différentes peuvent porter le même nom**
- Le compilateur choisit une fonction grâce à la liste (typée) de ses arguments au moment de son appel
- On peut aussi surcharger des opérateurs (*, +, etc...)
- La signature d'une fonction est composé de son nom et de la liste de ses arguments. **Attention, le type de la valeur de retour n'est pas un élément discriminant !**

```

int somme(int n1, int n2) { return n1+n2; }
double somme(int n1, int n2) { return (double)(n1+n2); }
// ERROR : new declaration 'double somme(int, int)'
// ERROR : ambiguates old declaration 'int somme(int, int)'
double somme(double d1, double d2) { return d1+d2; } // OK
int somme(int n1, int n2, int n3) { return n1+n2+n3; } // OK

typedef struct { double x; double y; } Point;

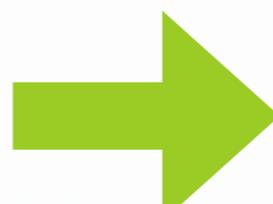
Point somme(const Point& p1, const Point& p2) {
    Point p;
    p.x = p1.x + p2.x; p.y = p1.y + p2.y;
    return p;
} // OK

int main() {
    Point p = {3.0,1.0}, q = {0.2,2.7};
    affiche(p, 0, "p");

    cout << "3+4 = " << somme(3,4) << endl;
    cout << "2.6+7.2 = " << somme(2.6,7.2) << endl;

    Point r = somme(p,q);
    affiche(r, 0, "p+q");
    return 0;
}

```



```

$ p = (3.000000,1.000000)[0]
$ 3+4 = 7
$ 2.6+7.2 = 9.800000
$ p+q = (3.200000,3.700000)[0]

```



Surcharge de fonctions en C++

20



Surcharge d'opérateurs en C++ (voir aussi chapitre dédié)

```
int somme(int n1, int n2) { return n1+n2; }
double somme(int n1, int n2) { return (double)(n1+n2); }
// ERROR : new declaration 'double somme(int, int)'
// ERROR : ambiguates old declaration 'int somme(int, int)'
double somme(double d1, double d2) { return d1+d2; } // OK
int somme(int n1, int n2, int n3) { return n1+n2+n3; } // OK
```

```
typedef struct { double x; double y; } Point;
```

★

```
Point operator+(const Point& p1, const Point& p2) { <-----
    Point p;
    p.x = p1.x + p2.x, p.y = p1.y + p2.y;
    return p;
} // OK
```

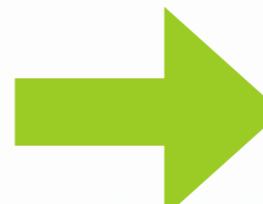
```
int main() {
    Point p = {3.0,1.0}, q = {0.2,2.7};
    affiche(p, 0, "p");
```

on remplace le nom de la fonction somme par le mot clé operator suivi du symbole + de l'opérateur d'addition

```
cout << "3+4 = " << somme(3,4) << endl;
cout << "2.6+7.2 = " << somme(2.6,7.2) << endl;
```

★

```
Point r = p+q;
affiche(r, 0, "p+q");
return 0;
}
```



```
$ p = (3.000000,1.000000)[0]
$ 3+4 = 7
$ 2.6+7.2 = 9.800000
$ p+q = (3.200000,3.700000)[0]
```

on somme directement des structures grâce à l'opérateur +

21



Spécificités du C++

> Entrées / Sorties

- **En C** : on utilise des fonctions d'E/S standard (**scanf**, **printf**...)
- **En C++** : on utilise des opérateurs d'E/S
 - << pour envoyer des valeurs dans un **flot de sortie**
 - >> pour extraire des valeurs d'un **flot d'entrée**
- Un flot peut être défini à partir d'un fichier, d'une entrée/sortie standard, d'un buffer de caractères etc.
 - **cin / cout** : entrée / sortie standard (éq. à **stdin / stdout**)
 - **cerr / clog** : sorties standard d'erreur
 - les principaux types standards sont reconnus (**char**, **float**, **int** etc.)



Entrées/Sorties en C++

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

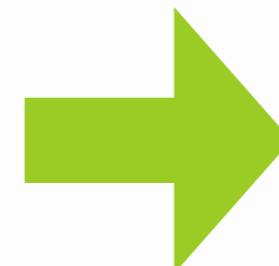
```
int main() {
    int i=123;
    float f=1234.567
    char str[]="Bonjour\n", rep;

    cout << "i=" << i << "  f=" << f << " str=" << str;
    cout << "i = ? ";
    cin >> i;
    cout << "f = ? ";
    cin >> f;
    cout << "rep = ? ";
    cin >> rep;
    cout << "str = ? ";
    cin >> str;
    cout << "i=" << i << "  f=" << f << " str=" << str << endl;
    cout << "adresse de str = " << (void*)str << endl;

    return 0;
}
```

utilisation de l'espace de nom std :
permet d'accéder directement aux flots
`std::cout` et `std::cin`

les opérateurs << sont associatifs à droite : on peut concaténer les E/S de gauche à droite



```
> i=123  f=1234.57
> i = ? 54
> f = ? 2.8
> rep = ? z
> str = ? C++ is easy
> i=54  f=2.8 str=C++
> adresse de str = 0xbfffffa9b
```

23

endl est un flot particulier qui insère un retour chariot et vide le buffer (comme `fflush`)

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

...

Vecteur

état :

dimension
liste des coefficients

comportement :

ajouter un vecteur
produit vectoriel
mult. / matrice

...

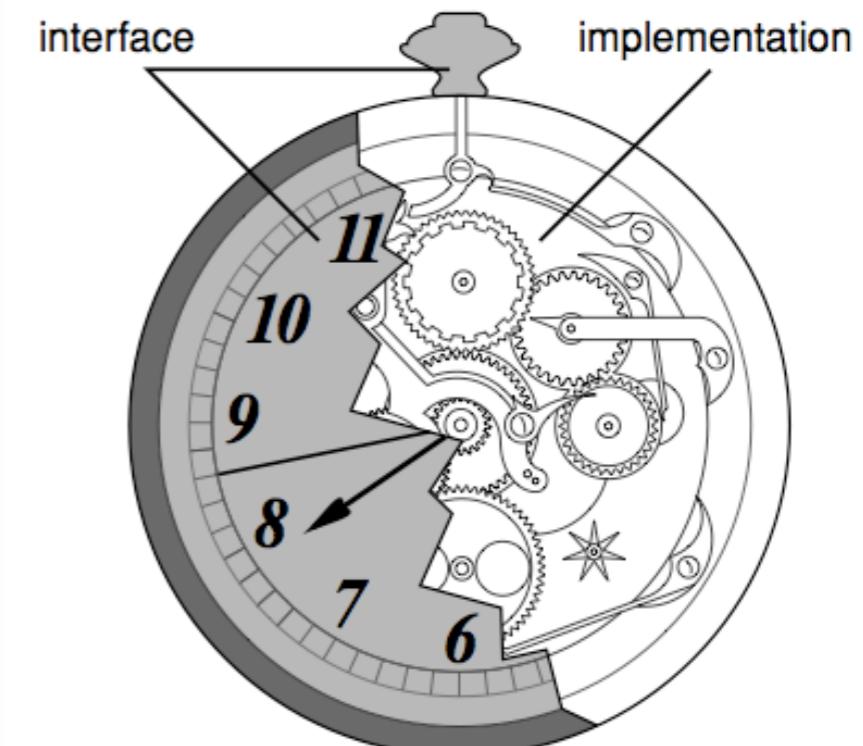
$$a = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$



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

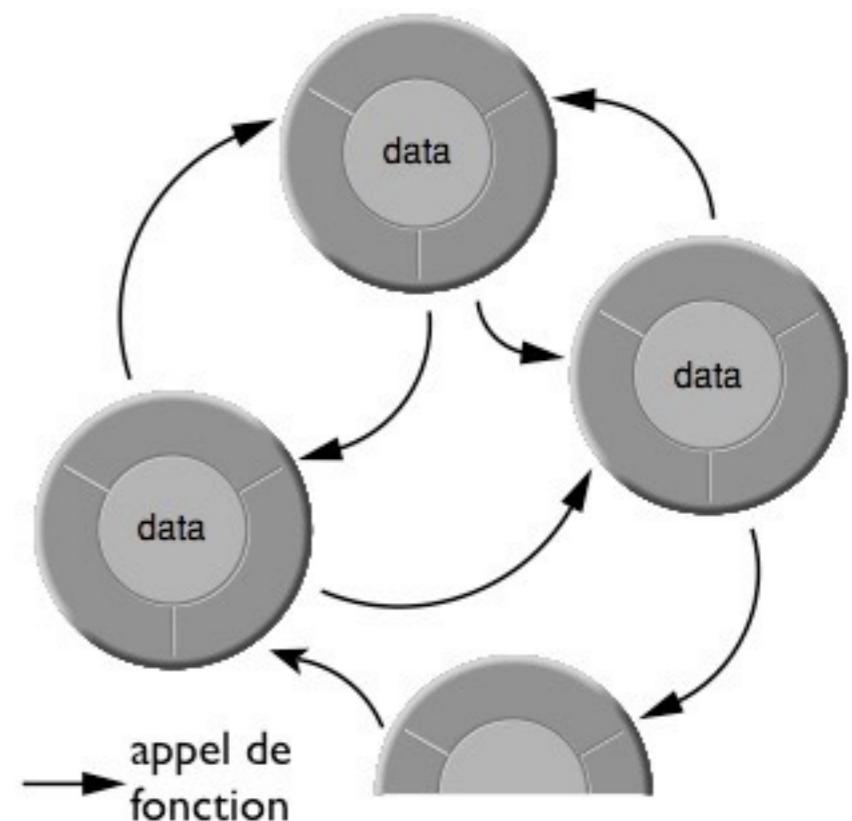
fonctions :
mult(réel *r*) {
 pour *i* de 1 à *dim* faire
 coeffs[*i*] <- (*coeffs*[*i*] × *r*)
 }
....



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 :

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



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

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

Implémentation n°1

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

Implémentation n°2

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

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

coeff(entier i) : réel;

...

Plusieurs implémentations possibles pour une même interface

attributs :

réel x, y, z;

méthodes :

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

...

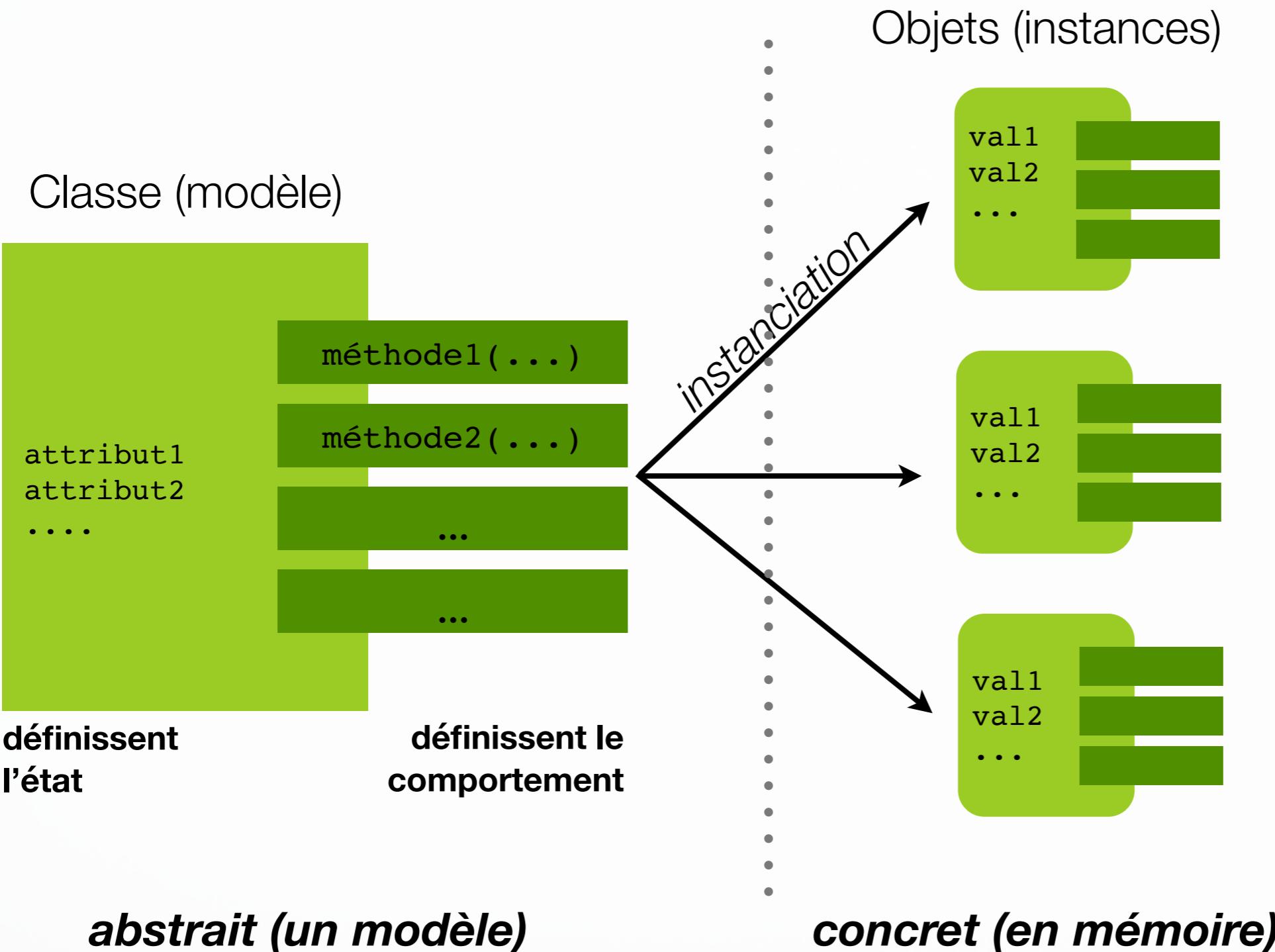
attributs :

réel a[3];

méthodes :

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

Notion de classe Récapitulatif



36

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
```

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

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

Instanciation automatique de la classe Car

```
// 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;  
}
```

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



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

en sortie

> clio

40

Instanciation dynamique de la classe Car

```
// 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 ->

Les classes
en C++

> Exemple

41

Définition des méthodes de la classe Car



Les classes en C++

> Exemple

```
// 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

42

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**

43



Les classes
en C++

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)
}
```

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

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

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 chaîne de caractères
    void setName(const string& name);

    //attribue le 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);
```



```
void CarSetName(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();
}
```

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

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**

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)

I' 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»)

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

définition du constructeur `Car` : on initialise les attributs



Constructeurs

```
// 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;
}
```

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

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 {

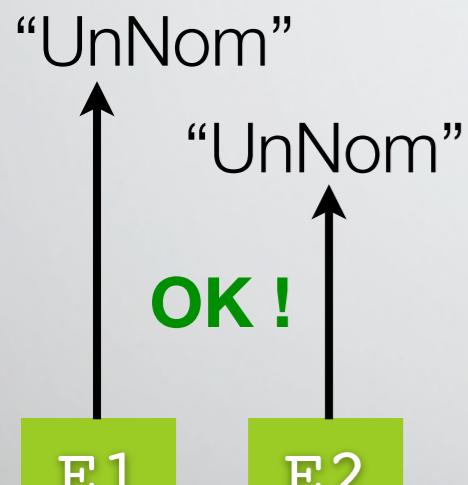
    string _name;
public:
    //méthodes publiques
    Car(const string& name="Voiture");

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

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

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



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 © Groupe Effei Nicolas Sicard - v4

Constructeurs

> Initialiser à la construction

■ Exemple de la classe Car

```
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) !  
}
```

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

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);
    }
}
```



Les classes
en C++

62

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

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

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

63

Les classes
en C++

Constructeurs / Destructeurs

> Visibilité

- **Constructeurs et destructeurs 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
};
```

on déclare un attribut de classe **public** de type **int** en le précédent du qualificatif **static**

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

on peut utiliser cet entier directement (prefixé 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 incrémenté le compteur _count
}

Car::~Car() {
    Car::_count--; // on décrémente le compteur _count
}
```

une nouvelle instance : on
incrémenté 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 qualifiées 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; }
};
```

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

on déclare/définit une méthode de classe inline qui retourne le nombre d'instances 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 Resource& 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
};


```

72

```

// 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++;
    }
}

```

73

```

// 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;
}

```

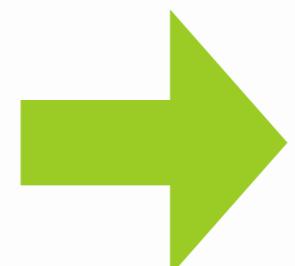
75

```
// 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`

```
#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);
};
```

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

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

Surcharges d'opérateurs

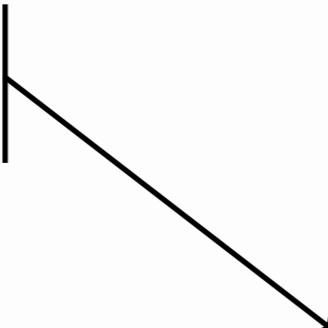
- Objectif : permettre l'utilisation plus intuitive d'une classe à partir d'opérateurs connus
- Exemple (`Vector3`)

```
Vector3 v, v1(1.1, 2.2, 3.3), v2(0.3, 0.4, 0.5);  
Matrix3 mat;
```

```
// sans les opérateurs  
v2.mult(2.0);  
v1.add(v2);  
mat.rotate(0z, M_PI/2.0);  
v = mat.mult(v1);
```

```
// avec les opérateurs  
mat.rotate(0z, M_PI/2.0);  
v = mat * (v1 + 2.0 * v2);
```

cette expression est bien plus
compacte et plus claire que la
séquence d'instructions précédente



Surcharge des opérateurs

> Considérations générales

- Tous les opérateurs du C++ peuvent être surchargés sauf :
 - `::, ., .., * , ?:, sizeof, typeid, static_cast, dynamic_cast, const_cast, reinterpret_cast`
- Il est **impossible de changer la priorité, l'associativité et l'arité des opérateurs** (ex : ++ unaire, + binaire, ?: ternaire et () n -aire)
- Il est **impossible de créer de nouveaux opérateurs**
- Il est préférable de respecter la sémantique habituelle de l'opérateur pour que son utilisation reste intuitive

Surcharge des opérateurs

> 2 approches différentes

▪ Surcharge des opérateurs interne

- on considère ces opérateurs comme des méthodes normales de la classe (déclarés à l'intérieur de la classe)
- syntaxe : `Type operatorOp(paramètres);`
- alors : `a Op b <=> a.operatorOp(b)`

▪ Surcharge des opérateurs externe

- les opérateurs sont en fait des fonctions définies en dehors (mais souvent amies) de la classe concernée
- syntaxe : `friend Type operatorOp(paramètres);`
- alors : `a Op b <=> operatorOp(a,b)`

Surcharge des opérateurs interne

> Affectations et arithmétiques

- Opérateurs : =, +=, ++, -=, /=, *=, +, -, *, /
- Avec cette syntaxe **le premier opérande est l'objet sur lequel s'applique l'opérateur**
 - les opérantes suivantes sont les paramètres de la fonction opérateur
 - ex : `v1 += v2 <=> v1.operator+=(v2)` //v1 modifié
 - ex : `v = v1 * v2 <=> v.operator=(v1.operator*(v2))`
- Les opérateurs =, +=, ++, -=, /= et *= doivent retourner par référence l'objet sur lequel il travaillent par : `return *this;`
- Les opérateurs +, -, * et /, doivent retourner une copie du résultat calculé

```

// fichier vector3.h

class Vector3 {
    double _coeff[3];

public:
    Vector3& operator= (const Vector3& v);

    Vector3& add(const Vector3& v);

    Vector3& operator+= (const Vector3& v);

    Vector3& operator+= (const double s);

};

    
```

// fichier main.cc

```

Vector3 v1, v2, v3;
...
v1 = v2 = (v3 += 2.0);
v1 *= 5.0;
    
```

les opérateurs d'affectation ou d'affectation étendue retournent une référence sur l'objet : **on donc peut les chaîner !**

maintenant, on peut utiliser les opérateurs d'affectation ou d'affectation étendue sur des vecteurs comme on en a l'habitude pour les types simples



// fichier vector3.cc

```
Vector3& Vector3::operator= (const Vector3& v)
{
    if (&v != this) {
        _coeff[0] = v._coeff[0]; _coeff[1] = v._coeff[1]; ...
    }
    return *this;
}
```

on ne réalise l'affectation que si la source est une instance différente

```
Vector3& Vector3::add(const Vector3& v)
{
    _coeff[0] += v._coeff[0]; _coeff[1] += v._coeff[1]; ...
    return *this;
}
```

```
Vector3& Vector3::operator+= (const Vector3& v)
{
    return this->add(v);
}
```

il ne faut pas oublier de retourner une référence sur l'objet courant pour assurer l'associativité de l'opérateur

```
Vector3& Vector3::operator+= (const double s)
{
    _coeff[0] += s; _coeff[1] += s; _coeff[2] += s;
    return *this;
}
```

85

Surcharge des opérateurs internes

> Remarques sur l'opérateur =

- Deux problèmes à traiter
 - cas de l'auto-affectation : `obj = obj`
 - l'affectation de deux objets quand ils contiennent des pointeurs peut nécessiter une allocation mémoire

```
class Car {
protected:
    char *_name;
public:
    Car(const char* desc);
    Car(const Car& exp);
    Car& operator=(const Car& c);

    const char* getName() const;
};
```

on ne traite l'affectation que si les deux objets ne sont pas un seul et même objet : **gain de temps !**

```
Car& Car::operator= (const Car& c) {
    if (&c != this) {
        if (_name != NULL) free(_name);
        _name = strdup(c._name);
    }
    return *this;
}
```

comme dans le cas du constructeur de copie, **on doit recréer une copie de la chaîne de caractères _name**

Surcharge des opérateurs internes

> Opérateurs ++ et --

- Opérateurs doubles : préfixes (++i, --j) ou suffixes (i++, j--)
 - pas de paramètre s'ils sont préfixes, un paramètre fictif s'ils sont suffixes

```
class Vector3 {  
protected:  
    double _coeff[3];  
public:  
    Vector3(double x, double y, double z, double w=1.0);  
  
Vector3 v(1.0), v2;  
  
v2 = ++v; ←  
v2.print();  
  
v2 = v++; ←  
v2.print();  
  
Vector3& operator++() {  
    ++_coeff[0]; ++_coeff[1]; ++_coeff[2];  
    return *this;  
}  
Vector3 operator++(int i) {  
    return Vector3(_coeff[0]++, _coeff[1]++, _coeff[2]++);  
};
```

on retourne une copie :
pouvez-vous dire pourquoi ?

Surcharge des opérateurs internes

> Opérateur []

- Deux formes (une **const**, l'autre non)

```
class Vector3 {  
    double _coeff[3];  
  
public:  
  
    double& operator[](const int i) { return _coeff[i]; } // (1)  
    double operator[](const int i) const { return _coeff[i]; } // (2)  
};  
  
void printVector(const Vector3& v) {  
    cout << "["<<v[0]<<","<<v[1]<<","<<v[2]<<"]" << endl; // appels de (2)  
}  
  
void initVector(Vector3& v) {  
    v[0] = v[1] = v[2] = 0; // appels de (1)  
}
```

ici, on ne peut pas modifier le contenu de v, c'est donc l'opérateur const (2) qui est utilisé

là, on modifie le contenu de v, c'est donc l'opérateur non const (1) qui est utilisé

Surcharge des opérateurs internes

> Opérateur ()

- Deux formes (une **const**, l'autre non)

```
class Matrix3 {  
    Vector3 _line[3];  
  
public:  
    Vector3& operator[](const int i) { return _line[i]; }  
  
    double& operator()(const int i, const int j) { return _line[i][j]; }      //(1)  
    double operator()(const int i, const int j) const { return _line[i][j]; } // (2)  
};  
  
int main() {  
    Matrix4 m;  
  
    m(2,2) = 1.0; // appel de (1) car on modifie l'état de la matrice  
    cout << m(1,0) << endl; // appel de (2)  
  
    m[1][2] = 0.0; // quel(s) opérateur(s) ???  
}
```

Surcharge des opérateurs externes

> Opérateurs arithmétiques

- Problème : la surcharge d'opérateurs en interne impose que l'objet concerné soit la première opérande
 - $v * 2.0$ (avec v de type `Vector3`) est possible mais pas $2.0 * v$
 - mais on peut utiliser une fonction `operator` amie (`friend`)

```
// fichier Vector3.h
class Vector3{
public:
    friend Vector3 operator*(const double s, const Vector3& v);
    ...
};

//fichier Vector3.cc
Vector3 operator*(const double s, const Vector3& v) {
    return Vector3(s*v._coeff[0], s*v._coeff[1], s*v._coeff[2]);
}
```

```
Vector3 v(1.0), v2;
v2 = 2 * v; // OK!
// <=> v2 = operator*(2.0,v);
```

Surcharge des opérateurs d'E/S

> Opérateurs << et >>

- On a vu qu'on peut utiliser l'opérateur << pour écrire sur un flot de sortie standard **cout** (**cout** << **x**;) et l'opérateur >> pour lire depuis un flot d'entrée standard **cin** (**cin** >> **x**;)
- Ces opérateurs assurent le transfert d'information ainsi que son éventuel formatage
- Les flots prédéfinis **cout** et **cin** sont en fait des objets (instances) de la classe **ostream** et **istream** respectivement
- Ces deux classes surchargent les opérateurs << et >> pour les principaux types de base (**int**, **float**, **double** etc.)
- Ces opérateurs retournent une référence au flot d'E/S, ce qui permet de l'appliquer plusieurs fois de suite : **cout** << "a=" << **x**;

Surcharge des opérateurs d'E/S

> Opérateurs << et >>

- On peut surcharger ces opérateurs pour n'importe quelle classe **classe** créée par l'utilisateur à condition de noter que :
 - le premier argument de ces opérateurs est un flot (`istream` ou `ostream`), **il n'est donc pas possible d'en faire une fonction membre de la classe utilisateur.** On pourra en revanche utiliser des fonctions indépendantes (le plus souvent *amies*)
 - **la valeur de retour doit obligatoirement être une référence au flot concerné**

```
[friend] ostream& operator<< (ostream& os, const Classe& obj);
```

```
[friend] istream& operator>> (istream& is, Classe& obj);
```

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

il faut inclure les définitions des types
`ostream` et `istream`

```
class Car {
    ...
public:
    //méthodes publiques
    Car(const string& name="Voiture");
    ...
}
```

les opérateurs d'E/S sont déclarés comme **fonctions amies** et définis dans le fichier source de la classe

```
friend ostream& operator<<(ostream& os, const Car& aCar);

friend istream& operator>>(istream& is, Car& aCar);

};

//fichier car.cc
ostream& operator<<(ostream& os, const Car& aCar) {
    os << aCar.getName();
    return os;
}

istream& operator>>(istream& is, Car& aCar) {
    is >> aCar._name;
    return is;
}
```

ici exp est ‘`const`’ car il ne sera pas modifié par l'opérateur de sortie

ici exp **n'est pas** ‘`const`’ car il sera modifié par l'opérateur d'entrée

```
int main (int argc, char * const argv[]) {  
  
    Car aCar("Twingo"); // on crée une instance de la classe Car  
  
    aCar.print(); // on affiche son nom  
  
    cin >> aCar;  
    cout << "aCar = " << aCar << endl;  
  
    Car *aCarPtr;  
    aCarPtr = new Car("Picasso");  
  
    cout << "*aCarPtr = " << *aCarPtr;  
  
    return 0;  
}
```

une Car est désormais transmissible
à un flux d'E/S (ostream ou istream)
comme tout autre type standard (int,
float, string...)

en sortie

> Twingo
< Clio
> aCar = Clio
> *aCarPtr = Picasso

saisi par l'utilisateur

94

Surcharge d'opérateurs

> Opérateurs de conversion

- **Conversions explicites** (opérateur `cast`) comme en C
 - ex : `int n; double z; ... z=double(n); ... n=int(z)`
- **Conversions implicites** mises en place par le compilateur selon le contexte, à plusieurs niveaux :
 - dans les affectations : conversion forcée dans le type de la variable réceptrice
 - dans les paramètres de fonction : conversion forcée dans le type du paramètre déclaré dans le prototype
 - dans les expressions : pour chaque opérateur, il existe des règles de conversion précises entre les différents arguments

Surcharge d'opérateurs

> Opérateurs de conversion

- **On peut définir des conversions pour des classes créées par l'utilisateur**
- Deux sortes de conversions “utilisateur” en C++ :
 - la conversion d'un type prédéfini vers le type classe en question. On va créer pour cela un **constructeur de conversion**
 - la conversion du type classe vers un type prédéfini (ou défini préalablement). Ce type de conversion sera effectuée par des **fonctions de conversion**

Opérateurs de conversion

> Constructeur de conversion

- Défini dans une classe A, un constructeur avec un seul argument de type T permet de réaliser une conversion d'une variable de type T vers un objet de la classe A

```
class Vector3 {  
    double _coeff[3];  
  
public:  
    Vector3(double s=0.0);  
    ...  
};  
  
void f(Vector3 v) { ... }  
  
int main() {  
    Vector3 v2, v1 = 0.5;           // <=> Vector3 v(0.5);  
    v2 = 2.7;                      // OK ! <=> v2 = Vector3(2.7)  
    f(2.7);                        // OK ! <=> f(Vector3(2.7));  
}
```

lors d'une affectation ou d'un passage de paramètre, **le constructeur de conversion sera appelé implicitement s'il existe**

Opérateurs de conversion

> Fonction de conversion

- Une fonction de conversion est une méthode qui effectue une conversion vers un type T
- Elle est nommée `operator T()` et **n'a pas de type de retour**
- **Elle doit (quand même !) retourner une valeur du type T**

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

class Vector3 {
    double _coeff[3];
public:
    Vector3(double s=0.0) {
        _coeff[0] = _coeff[1] = _coeff[2] = 0.0;
    }
    Vector3(double x, double y, double z) {
        _coeff[0] = x; _coeff[1] = y; _coeff[2] = z;
    }

    operator double() { //opérateur de conversion vers un double
        cout << "Cast double() pour un vecteur" << endl;
        return _coeff[0]; // retourne le premier coefficient
    }
};

void f(double d) { cout << "appel de fonction avec arg = " << d << endl; }

int main() {
    Vector3 v(2.1,3.7,1.8);
    f(7.2);
    f(v);
    return 0;
}
```

ici, un **transtypage du vecteur v vers le type double est réalisé implicitement** par le compilateur grâce à la méthode `operator double()` de la classe `Vector3`

en sortie

- > appel de fonction avec arg = 7.2
- > Cast double() pour un vecteur
- > appel de fonction avec arg = 2.1

Héritage, héritage multiple

- L'héritage est un des fondements de la POO
 - permet de réutiliser, de spécialiser et d'étendre des classes
- On définit une nouvelle **classe dérivée** à partir d'une **classe existante de base**
 - la classe dérivée hérite des propriétés (attributs et méthodes) de la classe de base
- Le C++ autorise l'**héritage multiple** : une classe peut hériter de plusieurs classes de base



Modèle objet > Composition & Agrégation

- **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 !!





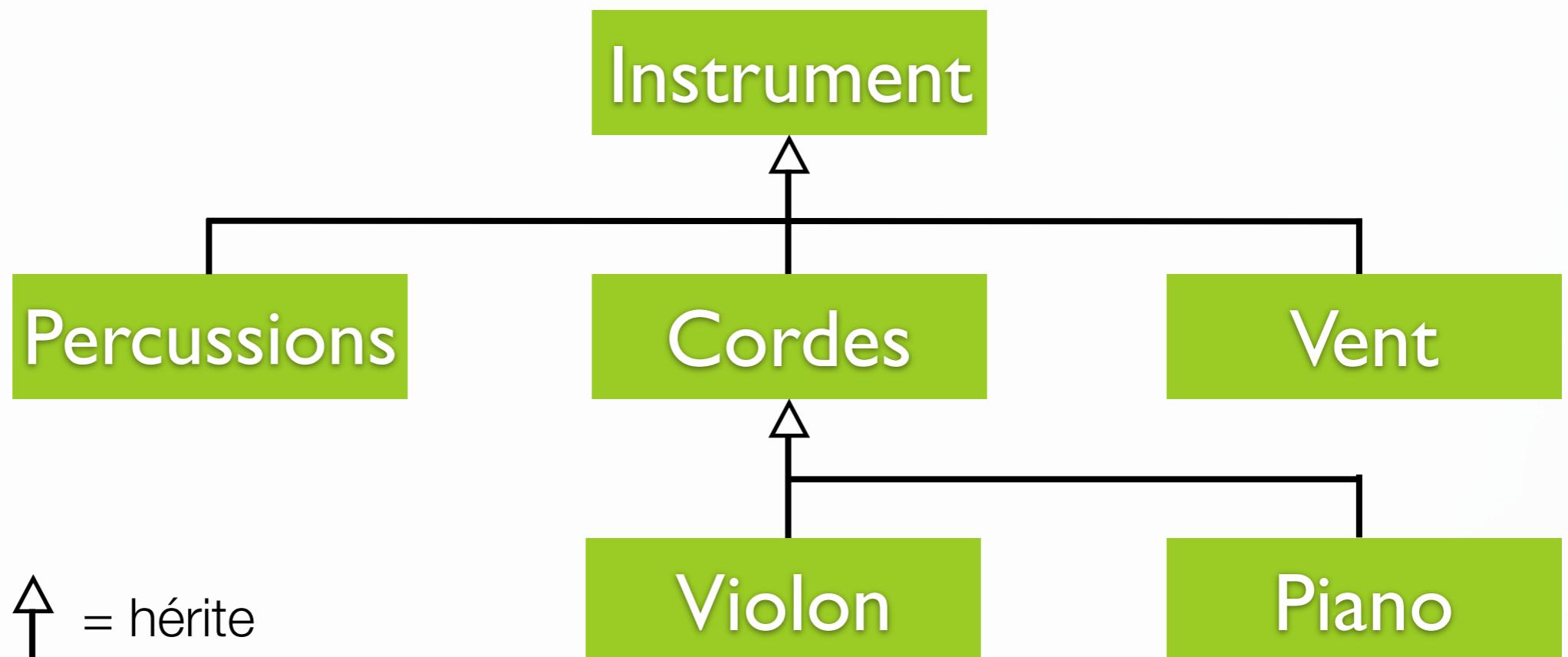
Modèle objet

> Héritage (spécialisation)

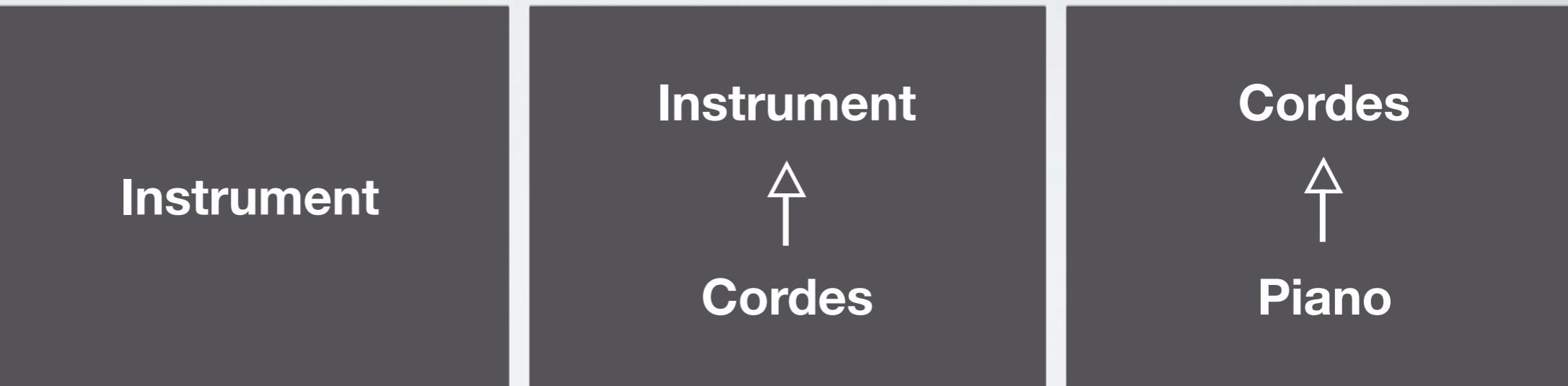
- Propriété qui permet de construire de nouvelles classes d'objet à partir de classes existantes
 - la classe existante est appelée **classe de base**
 - la nouvelle classe est appelée **classe dérivée**, elle **hérite de la structure (attributs) et du comportement (méthodes)** de la classe de base
- 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



Structure interne

string *nom*, *marque*;

Interface

init(string *nom*);

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

Structure interne

string *nom*, *marque*;
réel *cordes*[];

Interface

init(string *nom*);

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

setCordes(réel *r*[]);

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

setCordes = nouveau

↑ = hérite (*inherits*)

© Groupe Efri

Nicolas Sicard - v4

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)

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

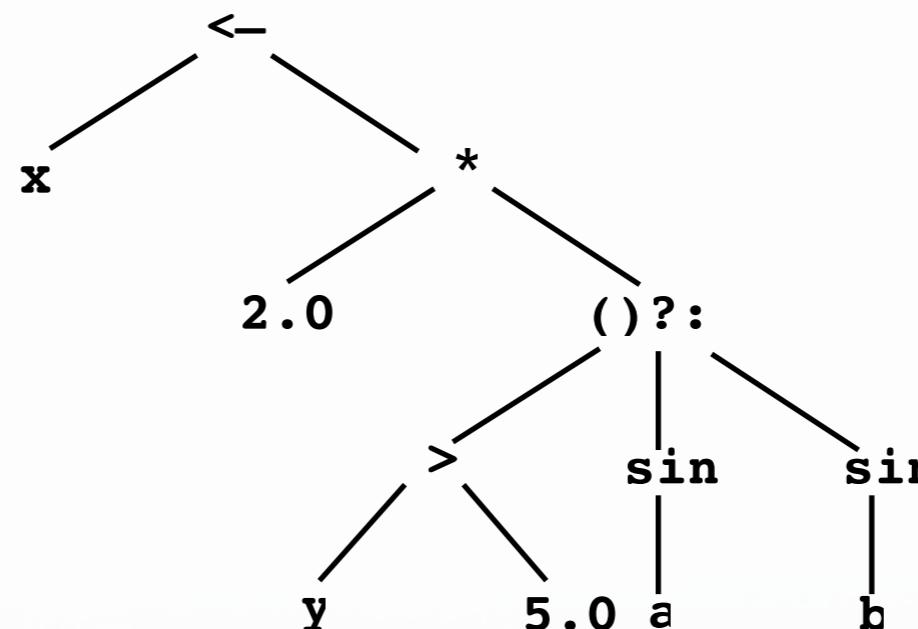
> Exemple

- 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

109



Modélisation objet de l'interpréteur

> Objets

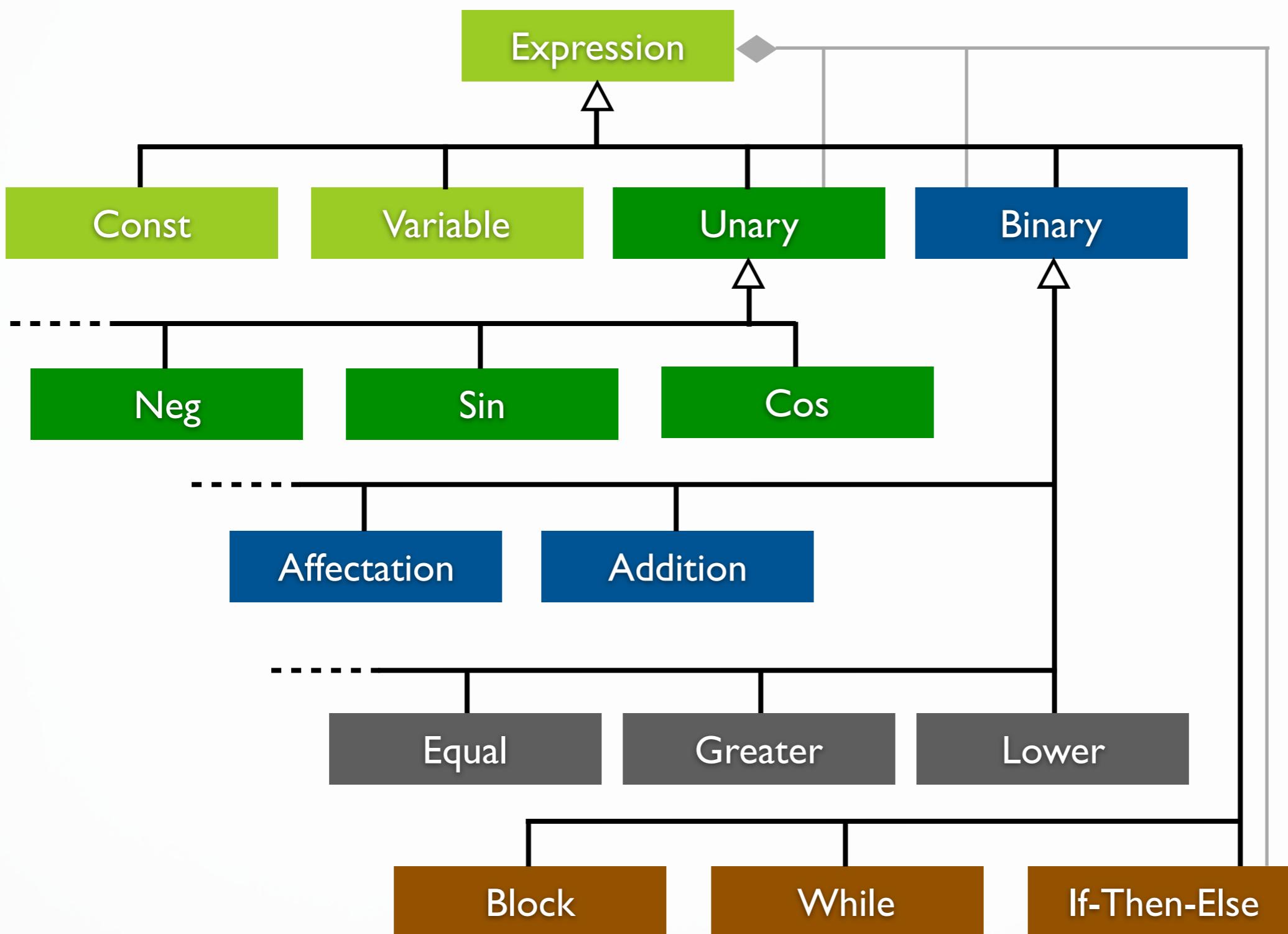
Cos...	Addition...	Affectation
- une représentation textuelle - une sous-expression	- une représentation textuelle - une sous-expression	- une représentation textuelle - une variable - une sous-expression
- initialiser - évaluer (cosinus) - obtenir représentation textuelle / afficher	- initialiser -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

110



Modélisation objet de l'interpréteur

> Diagramme de classes



111



Expressions arithmétiques

> Classe mère Expression

accesseur, mutateur
et méthode pour
l'affichage

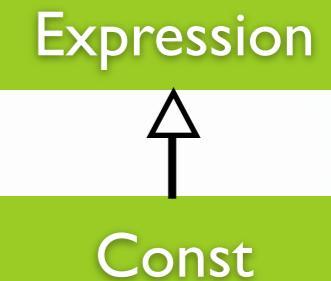
```
class Expression {  
    string _description;  
public:  
    Expression(const string& description="");  
  
    string getDescription() const;  
    void setDescription(const string& description);  
    void print() const { cout << getDescription(); }  
  
protected:  
    static unsigned long _count;  
public:  
    static unsigned long getInstanceCount();  
};
```

représentation textuelle d'une
expression arithmétique

attributs et méthodes de
classe pour compter le
nombre d'instances de la
classe Expression

Héritage

> Héritage simple



- On souhaite créer une classe fille **Const**, qui nous permet de représenter un expression constante (valeur numérique)
- une constante est une expression : on hérite donc de la classe de base **Expression**

```
// fichier const.h
#include "expression.h"
```

```
class Const : public Expression {
protected:
    double _value;
public:
    Const(const double value=0.0);
    ~Const();
    double eval() const; // retourne sa valeur
};
```

le mot public signifie que **l'héritage est public** : l'accessibilité des membres de la classe de base est conservée à l'identique

```

class Expression {
    string _description;
public:
    Expression(const string& description="");

    string getDescription() const;
    void setDescription(const string& description);
    void print() const;

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

La classe **Const** est une classe dérivée de **Expression**



```

class Const : public Expression {
protected:
    double _value;

public:
    Const(const double value=0.0);
    ~Const() {}

    double eval() const;
};
  
```

les attributs et méthodes hérités n'apparaissent pas dans la définition de la classe fille mais elles **sont implicitement disponibles**

114

la méthode **eval()** est une nouvelle méthode

Héritage

> Constructeurs / destructeurs

- Si la classe **B** dérive de la classe **A**
 - il faut **d'abord initialiser A** (en utilisant le bon constructeur) **puis compléter l'initialisation des attributs spécifiques de B** : le constructeur de **A** (soit par défaut, soit spécifié) est appelé avant celui de **B**

```
B::B(int i, float f) : A(), _attr1(i), _attr2(f) {}
```
 - on peut choisir le constructeur de la classe de base **A** avant l'initialisation des attributs de **B**

```
B::B(int i, float f) : A(i), _attr2(f) {}
```
 - **le destructeur de A est automatiquement appelé après le destructeur de B** (inutile de le spécifier)

Définition de la classe Const

```
// fichier const.h

class Const : public Expression {
protected:
    double _value;
public:
    Const(const double value=0.0);

    double eval() const;
};
```

```
// fichier const.cc
```

```
Const::Const(const double value) : Expression("CSTE"), _value(value)
{
    // rien : inutile de faire value = _value
}

double Const::eval() const { return _value; }
```

les méthodes héritées sont les méthodes publiques ou protégées de la classe Expression, donc déjà définies

on appelle un constructeur de la méthode de base Expression pour initialiser les attributs hérités

on initialise les nouveaux attributs de la classe Const

Utilisation de la classe Const

Héritage,
héritage multiple
> Exemple

```
// fichier main.cc

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

int main (int argc, char * const argv[])
{
    Const cste(12.3);

    cout << "description = " << cste.getDescription() << endl;
    cout << "value = " << cste.eval() << endl;
    cste.print();
    return 0;
}
```

en sortie

> **description** = CSTE
> **value** = 12.3
> CSTE

description vient par héritage
de la classe Expression

117

Héritage

> Redéfinition de méthode

- Exemple : modifier getDescription pour une constante

```
// fichier const.h

class Const : public Expression {
protected:
    double _value;
public:
    Const(const double value=0.0);

    double eval() const;
    string getDescription() const {
        return double2string(_value);
    }
};
```

on redéfinit la méthode
getDescription() de la
classe Expression avec la
même signature : elle retourne
alors la valeur constante sous
forme de string

Utilisation de la classe Const

Héritage,
héritage multiple
> Exemple

```
// fichier main.cc
```

```
#include <iostream>
#include "expression.h"

int main (int argc, char * const argv[])
{
    Const cste(12.3);

    cout << "description = " << cste.getDescription() << endl;
    cout << "value = " << cste.eval() << endl;
    cste.print();

    return 0;
}
```

en sortie

```
> description = 12.3
> value = 12.3
> CSTE
```

on obtient bien maintenant
la valeur de la constante

mais print affiche toujours CSTE
sauriez-vous dire pourquoi ?

119

Chemin d'appels pour print()

Héritage,
héritage multiple
> Exemple

la méthode print appelée est celle de la classe **Expression** : elle appelle donc la méthode **print()** de la même classe, qui affiche CSTE

Const cste(12.3);
cste.print();

```
class Expression {  
    string _description;  
public:  
    Expression(const string&);  
  
    string getDescription() const;  
    void print() const;  
};
```

«CSTE»

```
class Const {  
    double _value;  
public:  
    Const(const double);  
  
    string getDescription() const;  
};
```

string(_value)



120

Héritage

> Constructeurs de copie

- Si la classe **B** dérive de la classe **A**, trois cas se présentent :
 - aucun constructeur de copie n'est défini dans **B** : les attributs spécifiques de **B** sont copiés membre à membre et la "partie" héritée de **A** est traitée comme un membre de type **A**. Le constructeur de copie de **A** (défini par l'utilisateur, sinon par défaut) sera appelé !
 - **B([const] B& x)** : il y a appel au constructeur par défaut de **A** défini par l'utilisateur (s'il n'existe pas il y a erreur de compilation). Il ne faut pas oublier d'affecter les attributs de la partie **A** avec ceux de l'objet copié.
 - **B([const] B& x) : A(. . .)** : il y a appel au constructeur de **A** spécifié.

Constructeurs de copie

```
// fichier const.h
class Const : public Expression {
protected:
    double _value;
public:
    Const(const Const& c) : Expression(c), _value(c._value) {}

    string getDescription() const;
    double eval() const;
};
```

> Exemple

```
// fichier main.cc
int main (int argc, char * const argv[])
{
    Const c1(12.3);
    cout << "c1 = " << c1.eval() << endl;

    Const c2(c1);
    cout << "c2 = " << c2.eval() << endl;

    Expression exp(c2);
    cout << "exp = " << exp.getDescription() << endl;

    return 0;
}
```

c est passé en paramètre du constructeur de copie
Expression car c'est aussi une Expression !

on initialise c2 comme une copie de c1 : c'est
le constructeur de copie qui est appelé

en sortie

> c1 = 12.3
> c2 = 12.3
> exp = CSTE

122

Héritage

> Note à propos des **friends**

- **Lorsqu'une classe dérivée B (de A) déclare une fonction amie f(), celle-ci n'a pas les autorisations d'accès qu'ont les fonctions membres de B. En particulier, f n'a pas accès aux membres protégés de la classe de base A**
- **Une déclaration d'amitié ne s'hérite pas** : si B dérive de A et que la fonction f() est déclarée amie de A, f n'est pas amie de B !

```
class A {  
    int x;  
    friend void f(int t);  
};  
  
class B : public A {  
    friend void g(int u);  
    int y;  
};
```

```
void f(int t) {  
    B b;  
    b.x = 1; //OK : x attribut de A  
    b.y = t; //acces interdit  
}  
void g(int u) {  
    B b;  
    b.y = 2; //OK : y attribut de B  
    b.x = 3; //acces interdit  
}
```

Héritage

> Modes de dérivation

- Une dérivation peut être : **public**, **protected** ou **private**
 - ces modes changent les accès aux membres de la classe de base via la classe dérivée
 - par défaut (aucune mention du mode), C++ choisit le mode **private**
 - exemples :

```
class A { ... };  
  
class B : A { ... }; <=> class B : private A { ... };  
  
class C : protected A { ... };
```

Modes de dérivation

> class B : public A

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	OUI	public
protected	OUI	NON	protected
private	NON	NON	private

Modes de dérivation

> class B : protected A

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	NON	protected
protected	OUI	NON	protected
private	NON	NON	private

Modes de dérivation

> class B : private A

Attributs de la classe de base	Accès dans une classe dérivée	Accès pour un client de la classe dérivée	Nouveau statut dans la classe dérivée
public	OUI	NON	private
protected	OUI	NON	private
private	NON	NON	private

Modes de dérivation

> Ajustements d'accès

- Lors d'un héritage protégé ou privé, **on peut préciser que certains membres de la classe de base conservent leur mode d'accès dans la classe dérivée**
- attention, cela **ne permet pas d'augmenter ou de diminuer la visibilité** d'un membre de la classe de base, seulement de le conserver

```
class A {  
public:  
    void f1();  
    void f2();  
protected:  
    void f3();  
    void f4();  
};
```

```
class B : private A {  
public:  
    A::f1(); //f1() reste public dans B  
    A::f3(); //ERROR : un membre protégé ne peut devenir public  
protected:  
    A::f2(); //ERROR : un membre public ne peut devenir protégé  
    A::f4(); //f4() reste protégé dans B  
};
```

on indique que la méthode `f4` de `A` conserve son accès protégé malgré l'héritage “privé” de `A` par `B`

Ajustements d'accès

> Exemple

```
// fichier const.h
class Const : public Expression {
protected:
    double _value;
public:
    Const(const double value=0.0);
    Const(const Const& c);

    string getDescription() const;
    double eval() const;
protected:
    Expression::setDescription;
};
```

```
// fichier const.cc
int main (int argc, char * const argv[]) {
    Const c1(12.3);
    cout << "c1 = " << c1.eval() << endl;

    Const c2(c1);
    cout << "c2 = " << c2.eval() << endl;

    c2.setDescription("TOT0"); // ERROR : 'void Expression::setDescription(const
    std::string&)' is inaccessible
    return 0;
}
```

on ne peut plus appeler la méthode setDescription() appliquée à un objet de classe Const car elle est devenue protégée !

129

Héritage

> Conversion de type

- On peut **convertir implicitement** une instance (via objet, référence ou pointeur) d'une classe dérivée en une instance de la classe de base **si l'héritage est public**
- **L'inverse est interdit** (comment initialiser les membres de la classe dérivée ?)

```
Expression    exp, *p_exp;
Const        cste, *p_cste;
...
exp = cste;      // OK !
p_exp = p_cste; // OK !
cste = exp;      // ERROR !
p_cste = p_exp; // ERROR !
p_cste = (Const*)p_exp; // OK seulement si p_exp pointe vers Const !
```

Héritage

> Opérateur d'affectation

- Si la classe B dérive publiquement de la classe A
 - si B ne redéfinit pas l'opérateur =, l'affectation de deux objets de type B se déroule membre à membre en considérant que la partie héritée de A est un membre. Cette partie est alors traitée par l'opérateur d'affectation prévue dans A (redéfini ou par défaut)
 - si B redéfinit l'opérateur =, l'affectation de deux objets de B fait appel à cette fonction : il faudra donc prendre en charge tout ce qui concerne l'affectation d'objet de type A

Héritage

> Héritage multiple



- Dériver une classe à partir de plusieurs classes de base
 - pour chaque classe de base on peut définir un mode d'héritage
 - exemple : **C** dérive de **A** et de **B**

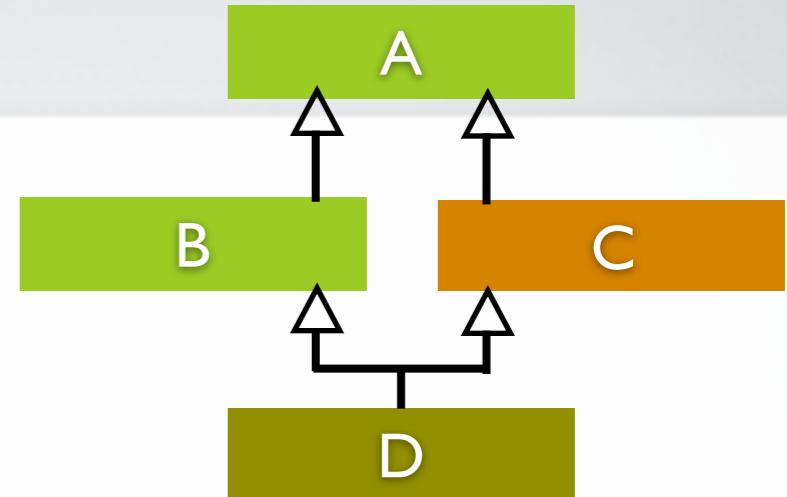
```
class C : protected A, public B {  
protected:  
    int    *_attr_c;  
  
public:  
    C(bool b, int *p_i) : A(), B(b), _attr_c(p_i) {}  
    ~C() {}  
  
    ...  
};
```

la classe **C** dérive à la fois de **A** (protégée) et publiquement de **B**

les constructeurs des membres dérivés sont appelés dans l'ordre d'héritage

Héritage

> Héritage virtuel



- Problème lorsqu'une classe hérite deux fois d'une autre

```
class A { int _x, _y; };
class B : public A { ... };
class C : public A { ... };
```

```
class D : public B, public C { ... };
```

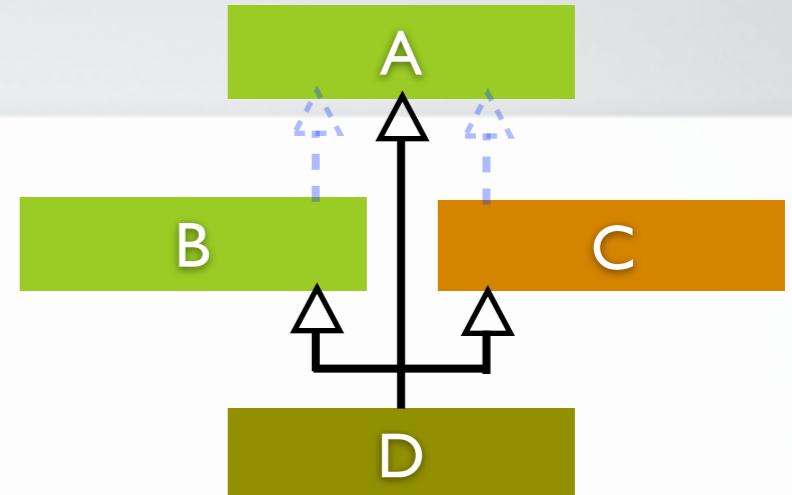
- Les membres de la classe A sont dupliqués dans tous les objets de classe D (problème du losange)
 - on peut n'incorporer qu'une seule fois les membres de A dans D

```
class B : public virtual A { ... };
class C : private virtual A { ... };
```

```
class D : protected B, public C { ... };
```

Héritage

> Héritage virtuel

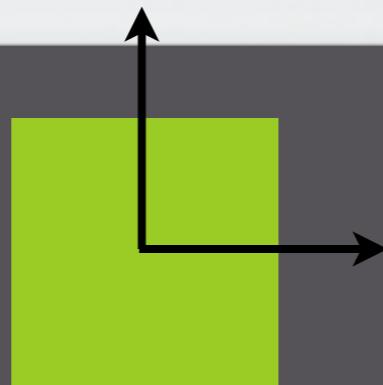


- Si A a été déclarée “virtuelle” dans B et C, un seul objet de A est construit : quels arguments transmettre au constructeur (de A) ?
- dans ce cas, on peut exceptionnellement spécifier dans le constructeur de D des informations destinées à A

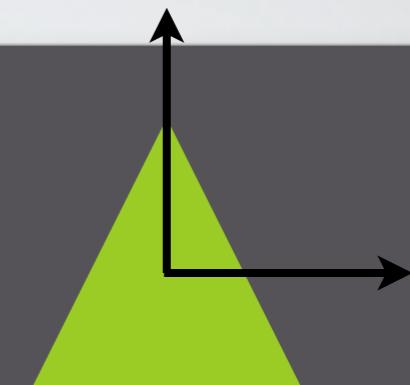
```
class A {  
    int _x, _y;  
};  
  
class B : public virtual A { double _z; };  
class C : public virtual A { };  
  
class D : public B, public C {  
public:  
    D(int a, int b, double z) : B(a,b,z), A(a,b) {}  
};
```

Polymorphisme et méthodes virtuelles

- L'**héritage** permet de réutiliser le code écrit pour la classe de base dans les classes dérivées
- Le **polymorphisme** permet de choisir dynamiquement (à l'exécution) une méthode en fonction de la classe effective de l'objet sur lequel elle s'applique
 - **le choix de la méthode s'appelle liaison dynamique**
- En C++, un pointeur peut recevoir l'adresse de n'importe quel objet de classe descendante mais la méthode choisie est systématiquement celle du type pointé
 - **par défaut, C++ réalise une liaison statique, déterminée au moment de la compilation**



Carré



Triangle (équilatéral)

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)  
}
```

Interface

```
dessiner()  
dessiner(couleur : entier)
```

Implémentation

x, y, L : réel

```
dessiner() {  
    allerÀ(x-L/2, y-Ltan(π/6)/2)  
    tracerVers(x+L/2, y-Ltan(π/6)/2)  
    tracerVers(x, y+L/2cos(π/6))  
    tracerVers(x-L/2, y-Ltan(π/6)/2)  
}
```

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

Polymorphisme

> Définition

- 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)

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

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) ?*

Liaison statique

> Exemple (1)

- Ajout de la méthode eval à la classe Expression

```
class Expression {  
    // attributs (privés par défaut)  
    string _description;  
public:  
    double eval() { return 0.0; } ← on suppose qu'une expression  
                                est évaluée à 0.0 par défaut  
    string getDescription() const;  
    void setDescription(const string& description);  
    void print() const;  
};
```

Classe Const : rappel

```
class Const : public Expression {  
  
protected:  
    double _value;  
  
public:  
    Const(const double value) : Expression("CSTE"), _value(value) {}  
  
    Const(const Const& c) : Expression("CSTE"), _value(c._value) {}  
  
    string getDescription() const {  
  
        return double2string(_value);  
  
    }  
  
    double eval() const { return _value; }  
};
```

la méthode eval() est redéfinie pour la classe Const

141

On crée la classe **Binary** (opérateur binaire générique) qui hérite de la classe **Expression** (car c'est une expression !)

```
class Binary : public Expression {  
  
protected:  
    Expression *_left, *_right; // deux sous-expressions  
  
public:  
    Binary(const Expression *l,  
           const Expression *r,  
           const string& desc="OPBIN") : Expression(desc), _left(l), _right(r) {}  
  
    string getDescription() const {  
        // retourne la concaténation «( _left OPBIN _right )»  
        return "(" + _left->getDescription() + " "  
               + Expression::getDescription() + " "  
               + _right->getDescription() + ")";  
    }  
};
```

on redéfinit la méthode `getDescription()` pour qu'elle retourne «(opleft OPBIN opright)» par concaténation des chaînes de caractères

142

On crée la classe **Plus** (addition) qui hérite de la classe **Binary** (car c'est un opérateur binaire !)

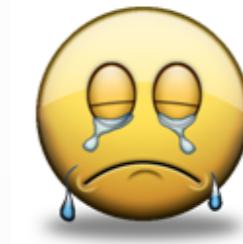
```
class Plus : public Binary {  
  
public:  
    Plus(const Expression *l,  
          const Expression *r) : Binary(l, r, "+") {}  
  
    double eval() const { return _left->eval() + _right->eval(); }  
};
```

```
int main () {  
  
    Const c1(12.3), c2(2.4);  
  
    Plus add(&c1, &c2);  
  
    cout << "add desc = " << add.getDescription() << endl;  
    cout << "add = " << add.eval() << endl;  
  
    return 0;  
}
```

évaluer une addition consiste simplement à sommer l'évaluation de ses deux sous-expressions gauche et droite

en sortie

> add desc = (CSTE + CSTE)
> add = 0



143

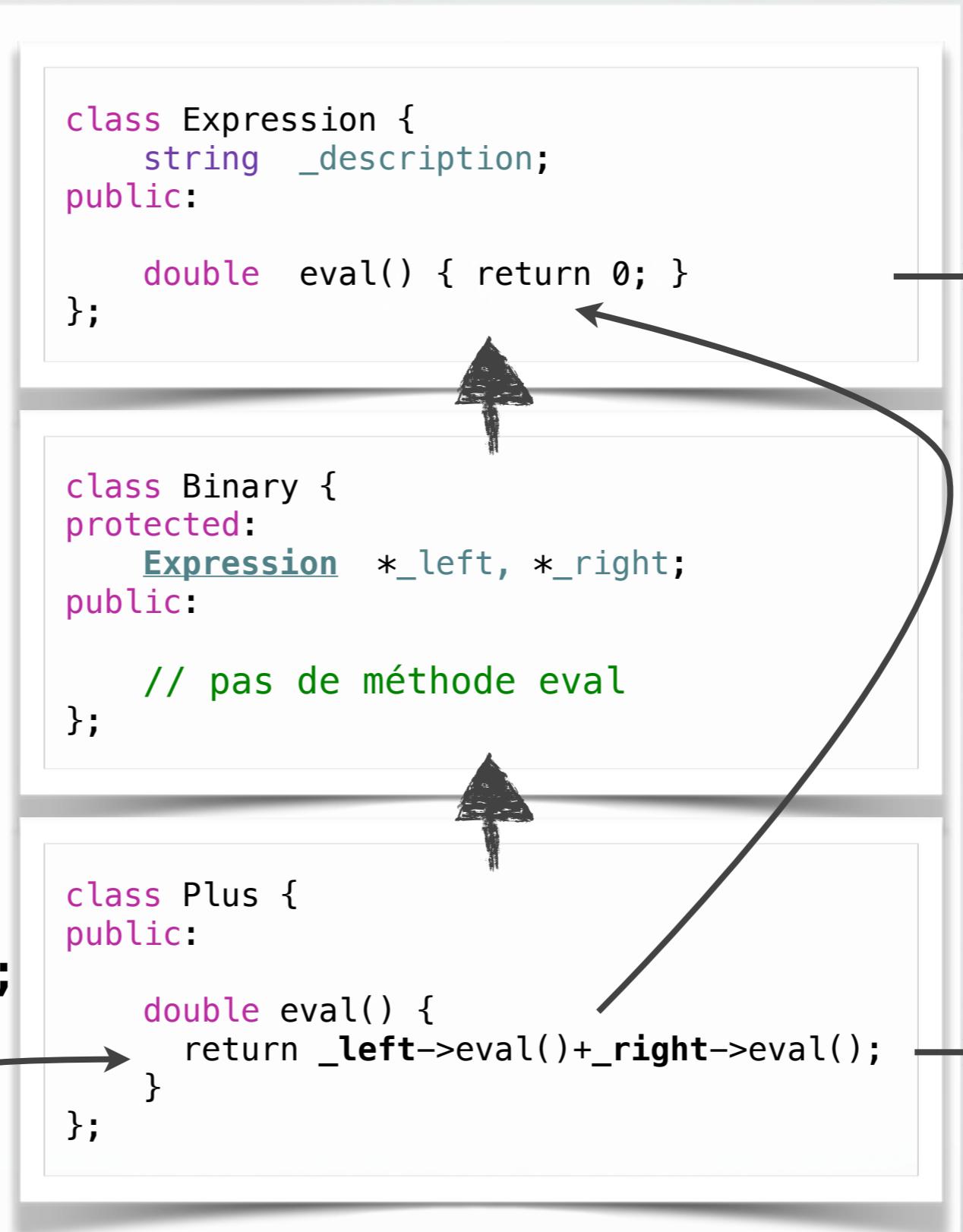
Pourquoi la somme est-elle égale à 0 et non à 14.7 ???

Liaison statique

> Exemple

_left et **_right** sont considérés comme des instances de la classe **Expression** et non de la classe **Const**

```
Const c1(12.3),c2(3.4);
Plus add(&c1,&c2);
add.eval();
```



0

0 + 0

144

Polymorphisme et méthodes virtuelles

> Liaison dynamique

- En C++, on peut demander une liaison dynamique en définissant des méthodes virtuelles
 - dans notre cas, il suffit de déclarer la méthode `eval()` virtuelle :

```
class Expression {  
    virtual double eval() const{  
        return 0.0;  
    }  
};
```
 - **ce mot clé précise au compilateur qu'il ne doit pas déterminer la méthode à appeler au moment de la compilation**
 - il mettra un dispositif en place **pour déterminer au moment de l'exécution quelle méthode eval() appeler, en fonction du type exact de l'objet**

Liaison dynamique

> Exemple

à l'exécution, la machine va chercher la méthode `eval()` de la classe réelle de `_left` et `_right`, c-à-d. `Const`, car `eval()` est virtuelle

```
class Expression {
    string _description;
public:
    virtual double eval() { return 0; }
};
```

0

```
class Binary {
protected:
    Expression *_left, *_right;
public:
    // pas de méthode eval
};
```

```
class Const {
    double _value;
public:
    double eval() { return _value; }
};
```

```
Const c1(12.3), c2(2.4);
Plus add(&c1,&c2);
add.eval();
```

```
class Plus {
public:
    double eval() {
        return _left->eval() + _right->eval();
    }
};
```

12.3 + 2.4

146

```
> add desc = ( CSTE + CSTE )
> add = 14.7
```



Polymorphisme et méthodes virtuelles

> Destructeurs

- Lorsqu'on définit une classe destinée à être dérivée, **il faut obligatoirement déclarer son destructeur virtuel**
 - cela permet d'assurer une liaison dynamique au niveau du destructeur
 - sans cela, la libération mémoire serait incomplète

```
Plus *p = new Plus(&exp1,&exp2);
```

```
p->print();
```

```
...
```

```
Expression *e = p;
```

```
...
```

```
delete e;
```

si le destructeur de Expression n'est pas déclaré virtuel, celui de Plus ne sera pas appelé à cette instruction !

```
class Expression {  
    virtual ~Expression();  
};
```

Polymorphisme et méthodes virtuelles

> Méthodes virtuelles pures

- Dans notre système, une **Expression** générique n'a pas de raison d'avoir une valeur !
 - la méthode `eval()` ne devrait rien retourner...
- Une solution consiste à déclarer la méthode `eval()` comme méthode virtuelle pure (**sans implémentation**)

```
// fichier expression.h

class Expression {
    ...
public:
    virtual double eval() const = 0;
};
```

en ajoutant '= 0' à la fin d'une déclaration de méthode, on indique qu'elle est virtuelle pure

Polymorphisme et méthodes virtuelles

> Classe abstraite

- **Une classe qui comporte au moins une méthode virtuelle pure est dite classe abstraite**
 - **on ne peut pas instancier une classe abstraite**
 - **MAIS on peut utiliser des pointeurs ou des références sur une classe abstraite**
- **Une méthode virtuelle pure (dans une classe de base abstraite) doit obligatoirement être redéfinie dans une classe dérivée.** Sans cela, la classe dérivée restera également abstraite

Classe abstraite

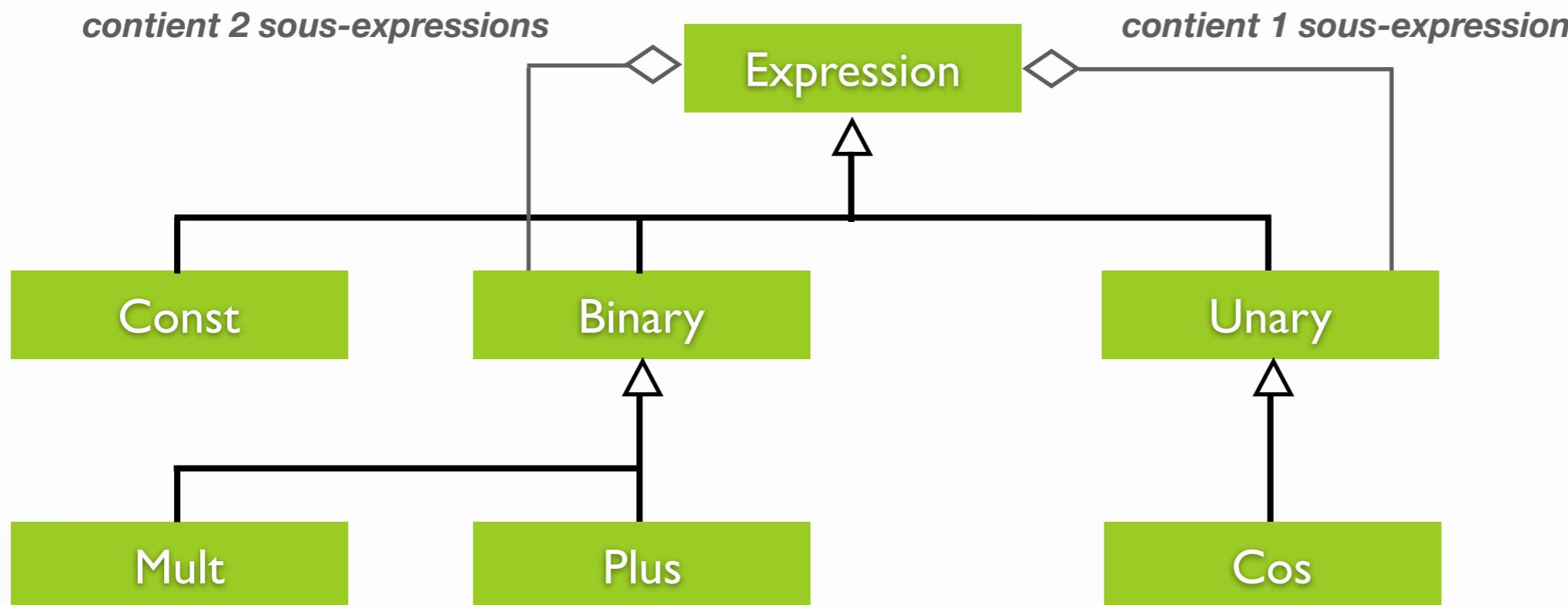
> Exemple (Expression)

- La classe Expression est abstraite

```
int main (int argc, char * const argv[]) {  
  
    Expression e("Titi");           // IMPOSSIBLE (classe abstraite)  
  
    e.setDescription("Toto");  
    e.print();  
  
    Expression *pe;                // OK : c'est un pointeur (pas une instance)  
    pe = new Expression("Chapeau pointu"); // NON (classe abstraite)  
  
    Const c1(0.5), c2(23.1); // OK : classe dérivée non abstraites  
  
    pe = new Plus(&c1,&c2); // OK : pe est un pointeur de classe mère  
  
    cout << pe->eval() << endl; // OK : quelle méthode eval() appelée ???  
}
```

Synthèse

> Diagramme de classe de l'interpréteur



151

Synthèse

> Classes Expression et Const

```
class Expression {  
    string _description;  
  
public:  
    Expression(const string& desc="EXP") : _description(desc) {}  
    Expression(const Expression& exp) : _description(exp._description) {}  
    virtual ~Expression() {}  
  
    virtual string getDescription() const { return _description; }  
    virtual void print() const { cout << this->getDescription() << endl; }  
  
    virtual double eval() const = 0;  
};
```

Polymorphisme
et méthodes
virtuelles

Expression est une classe abstraite : elle contient une méthode virtuelle pure (sans implémentation)

```
class Const : public Expression {  
protected:  
    double _value;  
  
public:  
    Const(const double value=0.0) : Expression("CSTE"), _value(value) {}  
    Const(const Const& c) : Expression(c), _value(c._value) {}  
  
    string getDescription() const { return string_from_double(_value); }  
    double eval() const { return _value; }  
};
```

152

la classe Const n'est pas abstraite : elle implémente la méthode virtuelle pure eval() de sa classe mère

Synthèse

> Classes Unary et Cos

Polymorphisme
et méthodes
virtuelles

```
class Unary : public Expression {  
protected:  
    const Expression * _op; // une opérande (sous-expression)  
public:  
    Unary(const Expression * op,  
          const string& desc = "UN") : Expression(desc), _op(op) {}  
    virtual ~Unary() { delete _op; }  
  
    string getDescription() const {  
        return Expression::getDescription() + "("  
            + _op->getDescription() + ")";  
    }  
};
```

la classe **Unary** est aussi abstraite car elle n'implémente pas la méthode **eval()** !

```
class Cos : public Unary {  
public:  
    Cos(const Expression * op) : Unary(op, "cos") {}  
  
    double eval() const { return cos(_op->eval()); }  
};
```

la méthode **eval()** de la classe **Cos** calcule le cosinus du résultat de la méthode **eval()** appliquée à son opérande

153

Synthèse

> Classes Binary, Plus et Mult

```
class Binary : public Expression {  
protected:  
    const Expression *_left, *_right; // deux opérandes (sous-expressions)  
public:  
    Binary(const Expression *l, const Expression *r,  
           const string& desc = "BIN") : Expression(desc), _left(l),_right(r) {}  
    virtual ~Binary() { delete _left; delete _right; }  
  
    string getDescription() const {  
        return "(" + _left->getDescription() + " "  
               + Expression::getDescription() + " "  
               + _right->getDescription() + ")";  
    }  
};
```

la classe **Binary** est aussi abstraite car elle n'implémente pas la méthode **eval()** !

```
class Mult : public Binary {  
public:  
    Mult(const Expression *l, const Expression *r) : Binary(l,r, "*") {}  
  
    double eval() const { return _left->eval() * _right->eval(); }  
};
```

Plus et Mult redéfinissent la méthode **eval()** : elles combinent les résultats de l'évaluation de leurs sous-expression

```
class Plus : public Binary {  
public:  
    Plus(const Expression *l, const Expression *r) : Binary(l,r, "+") {}  
  
    double eval() const { return _left->eval() + _right->eval(); }  
};
```

154

Synthèse

> Fonction principale

```
#include <iostream>
#include "expression.h"

int main (int argc, char * const argv[]) {
    Plus    exp(new Const(5.0),
               new Mult(new Const(2.0), new Cos(new Const(M_PI))));

    cout << "exp = " << exp.getDescription() << endl;

    cout << "exp.eval() = " << exp.eval() << endl;

    return 0;
}
```

en sortie

```
> exp = (5 + (2 * cos(3.14159)))
> exp.eval() = 3
```

Templates (patrons) & STL

- Les *templates* (patrons) fournissent un **mécanisme de production automatique de code** supplémentaire
 - ils permettent d'écrire des fonctions / algorithmes ou des classes conteneurs (piles, files, listes, arbres, *hash-table*...) pour n'importe quel type de données
 - ils permettent de **réutiliser des algorithmes stables, performants et éprouvés** pour différents types sans devoir réécrire le code (STL)
 - le **compilateur fabrique le code exécutable adapté à chaque type à partir du code patron**

Templates

> Patrons de fonctions

- Exemple : écrire la fonction de calcul du minimum de deux valeurs pour plusieurs types (`int`, `double`, voire `vector3...`)

```
int min(int a, int b) { return (a < b)? a : b; }
double min(double a, double b) { return (a < b)? a : b; }
```

- En C++ on peut simplement définir le patron de fonction

```
template <class T>
T min(const T& a, const T& b) { return (a < b)? a : b; }
```

on déclare un patron comportant un seul paramètre — type générique — que l'on appelle `T`, pour l'instant inconnu

le patron s'applique à une fonction de nom `min`, qui prend deux paramètres de type `T` et retourne une valeur de même type



Patrons de fonctions

> Instanciation du code

- Pour utiliser la fonction `min`, il suffit simplement de l'utiliser dans des conditions appropriées
 - deux arguments de même type
 - le type utilisé doit disposer d'un opérateur de comparaison <
- Par exemple, si on appelle `min(p, q)` avec `p` et `q` deux entiers `int`, le compilateur instanciera automatiquement la fonction `int min(int a, int b)`
- Le code d'une fonction patron ressemble au code d'une fonction classique mais en pratique sa définition sera souvent placée dans le fichier `.h` (car instancié chaque fois que nécessaire)

Patrons de fonctions

> Instanciation du code

- On peut aussi séparer en-tête et corps de fonctions dans les fichiers .h et .c

```
// myTemplateMin.h
template <class T> //déclaration du patron
T min(T a, T b);

// myTemplateMin.cc
#include "myTemplateMin.h"
template <class T> //définition du patron
T min(T a, T b)    // corps de la fonction
{
    return (a < b)? a : b;
}
```



Patrons de fonctions

> Remarques

- Un patron de fonction peut comporter plusieurs paramètres

```
template <class T, class U>
T somme(T a[], U op) { for (...) x=op.eval(x, a[i]); return x; }
```

- On peut surcharger des patrons de fonctions comme pour toute autre fonction

```
template <class T>
T min(T a, T b) { return (a < b)? a : b; }
```

```
template <class T>
T min(T a, T b, T c) { return min(min(a,b),c); }
```

- Un patron de fonction peut comporter également des paramètres non génériques ...

Patrons de fonctions

> Surcharge spécifique

- Que se passe-t-il si on utilise le type `char*` avec notre fonction template `min()` pour comparer des chaînes de caractères ?

on ne ferait que comparer des adresses (pointeurs) !

```
template <class T>
T min(T a, T b) { return (a < b)? a : b; }
```

- On peut surcharger des patrons de fonctions pour un type `T` particulier (dans le fichier `.cc`) :

```
#include <string.h>
char* min(char* a, char* b) {
    if (strcmp(a,b) < 0) ←
        return a;
    else
        return b;
}
```

la fonction `strcmp(a,b)` retourne une valeur négative si `a` est plus petit que `b` dans l'ordre lexicographique

Templates

> Classes génériques

- On peut définir de la même façon des patrons de classe, la générnicité s'applique alors à toute la classe :

```
template <class T>
class Vector3 {
    T _coeff[4];
public:
    Vector3(const T& s);

    T& operator[](int i);
    ...
}

template <class T>
T& Vector3<T>::operator[](int i) { return _coeff[i]; }

// déclaration de variables (fichier main.cc)
Vector3<int> v1(10);
```

Templates

> Classes génériques

- On peut même utiliser une valeur de type simple (`int`, `double`...) comme paramètre, par exemple pour dimensionner la taille d'un type générique lors de son instantiation :

```
template <class T, unsigned int size
```

Templates

> Classes génériques

- Ou comment être l'*ami* du patron ? ;-)

```
template <class T>
    class MaClasse {
        T val ;
    public:
        friend parle( );                                // fonction standard amie

        friend Copine<T> ;                            // patron ami lié
        friend fonctionCousine<T>();                  // patron ami lié
        friend Cousine::germaine(MaClasse <T>); // patron ami lié

        template <class Y>
            friend class SaClasse;                    // patron ami non lié
        template <class Y>
            friend void ParleA(MaClasse <Y>); // patron ami non lié
        template <class Y>
            friend void <Y>::g() ;                  // patron ami non lié
    };
```



Classes génériques

> Standard Template Library (STL)

- C++ inclut en standard (multi-plateforme) une bibliothèque de classes génériques très utilisées : la STL
 - listes, tableaux, tables de hachage, itérateurs et autres algorithmes génériques (recherche...)
- Permet d'unifier les définitions génériques les plus classiques comme les listes, les arbres *etc.*
- Permet de réduire fortement le temps de développement et de se concentrer sur la valeur ajoutée de son application
- Permet de bénéficier du travail testé et éprouvé de nombreux développeurs dans le domaine



Standard Template Library (STL)

> Conteneurs & Algorithmes

- Conteneurs : classes dont le rôle est de stocker d'autres objets

```
vector<int> v(3);           // déclare un vecteur de 3 éléments.  
  
v[0] = 7;  
v[1] = v[0] + 3;  
v[2] = v[0] + v[1];        // v[0] == 7, v[1] == 10, v[2] == 17
```

- On peut manipuler ces conteneurs via des algorithmes et outils classiques :

```
reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7  
  
int n = 12;  
v.resize(n); // redimensionne dynamiquement le vecteur
```

Classes génériques

> Standard Template Library (STL)

- **vector** : tableau redimensionnable dynamiquement
- **list, deque** : listes doublement chaînées
- **set, multiset** : ensemble d'éléments triés avec (**multiset**) ou sans doublon autorisé (**set**)
 - unions, intersections, insertions rapides...
- **map, multimap** : tableaux associatifs avec possibilité de définir un ordre particulier, strict (**map**) ou non strict (**multimap**)
- **hash_map, hash_multimap** : tableaux associatifs avec accès rapide (tables de hachage) mais sans ordre particulier



Standard Template Library (STL)

> Itérateurs

- Les itérateurs sont une **généralisation des pointeurs** pour des structures non alignées en mémoire :

```
list<double> l;           // liste vide

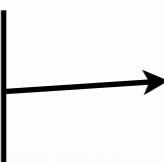
l.push_front(-2.7);      // insertion en tête
l.push_back(2.3);        // insertion en queue
l.push_front(5.9);       // insertion en tête

list<double>::iterator it = l.begin();

cout << *it << endl;    // 5.9
it++;
cout << *it << endl;    // -2.7

for (it = l.begin(); it != l.end(); it++) // 5.9, -2.7, 2.3
    cout << *it << ", ";
```

on parcourt la liste en
commençant au début et on
affiche chaque valeur





Standard Template Library (STL)

> Itérateurs

- On peut réutiliser le même algorithme pour parcourir un vecteur :

```
vector<double> v(3);           // vecteur de 3 éléments

v[0] = 5.9;
v[1] = -2.7;
v[2] = 2.3;

vector<double>::iterator it = v.begin();

cout << *it << endl;      // 5.9
it++;
cout << *it << endl;      // -2.7

for (it = v.begin(); it != v.end(); it++) // 5.9, -2.7, 2.3
    cout << *it << ", ";
```

on parcourt le vecteur en
commençant au début et on
affiche chaque valeur



Standard Template Library (STL)

> Itérateurs

- Mécanisme qui permet de découpler structures et algorithmes :

la fonction template `find()` cherche `value` dans un conteneur et retourne un itérateur (=`last` si `value` n'est pas présente)

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

...

```
list<double> l; // liste vide
```

...

```
list<double>::iterator il = find(l.begin(), l.end(), -2.7);
cout << *il << endl; // -2.7
```

```
vector<double> v(3); // vecteur
```

...

```
vector<double>::iterator iv = find(v.begin(), v.end(), -2.7);
cout << *iv << endl; // -2.7
```

on cherche la valeur `-2.7` dans **la liste** avec la fonction template `find()`

on cherche la valeur `-2.7` dans **le vecteur** avec la même fonction template `find()`

Standard Template Library

> Exemple

- On veut utiliser **un tableau associatif** (**map**) pour stocker les valeurs de nos variables et les retrouver rapidement par leur nom.
- Un tableau associatif permet de stocker des valeurs par couple (*key, value*). Pour nous la clé sera le nom de la variable et la valeur la valeur de la variable : ex : ("x", 2.7).
- On utilise la classe générique `map<class K, class V>` définie dans la STL.
- En outre, on va utiliser une classe standard du C++ (**string**) qui permet de manipuler aisément les chaînes de caractères.



Tableaux Associatifs

> Exemple

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

int main()
{
    map<char,int> mymap;
    map<char,int>::iterator it;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // afficher le contenu
    for (it = mymap.begin() ; it != mymap.end(); it++)
        cout << (*it).first << " => " << (*it).second << endl;

    return 0
}
```

en sortie

> a => 200
> b => 100
> c => 300

(source <http://www.cplusplus.com>)

172

```

map<string, double> var_tab;

var_tab["x"] = 2.6;
var_tab["y"] = 6.5e-2;
var_tab["z"] = -56.0;

cout << "x = " << var_tab["x"] << endl;
map<string, double>::iterator it;

it = var_tab.find("y");
if (it != var_tab.end()) {
    cout << it->first << " = " << it->second << endl;
} else {
    cout << "y does not exist" << endl;
}

it = var_tab.find("Y");
if (it != var_tab.end()) {
    cout << it->first << " = " << it->second << endl;
} else {
    cout << "Y does not exist" << endl;
}

```

la méthode **find** retourne un itérateur sur la paire («y», valeur) si elle existe, sinon elle retourne la valeur **end()**

l'itérateur **it** pointe sur une paire de type **pair<Key,Value>**, c.-à-d. une structure contenant deux champs : **first** (qui stocke la clé) et **second**, qui stocke la valeur



en sortie

> x = 2.6
> y = 0.065
> Y does not exist

173

```

// fichier variable.h
class Variable : public Expression {
protected:
    // tableau associatif des valeurs de toutes les variables
    static map<string,double> _memory;

public:
    Variable(const string& name) : Expression(name) {
        if (_memory.find(name) == _memory.end())
            _memory[name] = 0.0;                                | si la variable n'existe pas encore dans la
                                                               | mémoire on l'ajoute avec la valeur 0.0
    }
    Variable(const string& name, double val_init) : Expression(name) {
        _memory[name] = val_init;
    }

    double eval() const { return _memory[this->getDescription()]; }

    // méthodes de classe pour lire/écrire les valeurs de variables
    static double getValue(const string& name) {
        return _memory[name];
    }
    static void setValue(const string& name, double val) {
        _memory[name] = val;
    }
};

// fichier variable.cc
map<string,double> Variable::_memory;
  
```

ne pas oublier de définir
l'attribut de classe dans le
fichier source variable.cc

```

int main (int argc, char * const argv[]) {
  Mult *m = new Mult(new Const(2.0), new Cos(new Variable("x")));
  Plus exp(new Const(2.0), m);
  cout << "exp : " << exp.getDescription() << endl;
  for (unsigned i = 0; i <= 6; i++) {
    Variable::setValue("x", 2*i*M_PI/6.0);
    cout << "x = " << Variable::getValue("x")
    << " ; exp = " << exp.eval() << endl;
  }
  return 0;
}
  
```

en sortie

```

> exp : (2 + (2 * cos(x)))
> x = 0 ; exp = 4
> x = 1.0472 ; exp = 3
> x = 2.0944 ; exp = 1
> x = 3.14159 ; exp = 0
> x = 4.18879 ; exp = 1
> x = 5.23599 ; exp = 3
> x = 6.28319 ; exp = 4
  
```

on ajoute une méthode **dumpMemory()** qui permet d'afficher le contenu de la mémoire (toutes les variables)

```
// fichier variable.h
class Variable : public Expression {
protected:
    // tableau associatif des valeurs de toutes les variables
    static map<string,double> _memory;

public:
    ...
    // méthodes de classe pour lire/écrire les valeurs de variables
    static double getValue(const string& name);
    static void setValue(const string& name, double val);
    static void dumpMemory(); // affiche le contenu de la mémoire
};
```

```
// fichier variable.cc
map<string,double> Variable::_memory;

void Variable::dumpMemory() {
    map<string,double>::iterator it;

    cout << "Memory dump :" << endl;
    for (it = _memory.begin(); it != _memory.end(); it++)
        cout << "\t" << it->first << " = " << it->second << endl;
}
```

```

int main (int argc, char * const argv[])
{
  Mult      exp(new Const(2.0), new Cos(new Variable("x")));
  Variable::setValue("x", M_PI/3.0);
  cout << "exp = " << exp.eval() << endl;
  Variable   var_y("z", -11.7), var_z("y");
  Variable::dumpMemory();
  return 0;
}
  
```

en sortie

```

> exp = 1
> Memory dump :
> x = 1.0472
> y = 0
> z = -11.7
  
```

Exercices :

- 1/ Ajouter une méthode de classe pour vider la mémoire à la fin du programme.
- 2/ Modifier le programme de façon à ce qu'une variable disparaîsse de la mémoire s'il n'y a plus aucune référence vers elle...

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 associatif)
- Méthodes de classe
 - libération du *pool* (avec libération des objets restants)
 - ajout et retrait d’une instance dans le *pool*
 - recherche d’une instance par son nom

```

// fichier resource.h
#include <map>
#define POOL_SIZE    1024

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

protected:
    static map<string,Resource*> _pool; // tableau de ptr de ressources (idem)

    static void count() { return _pool.size(); } // retourne le nb d'instances de Resource
    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
};


```

179

```
// fichier resource.cc

map<string,Resource*>    Resource::_pool;

void Resource::releaseAllResources() {
    _pool.clear();
}

void Resource::registerResource(Resource *r) {
    if (r != NULL)
        _pool[r->_name] = r;
}

void Resource::unregisterResource(Resource *r) {
    if (r != NULL)
        _pool.erase(r->_name);
}
```

180

```

// fichier resource.cc (fin)

Resource* Resource::getResourceNamed(const string& name)
{
    map<string,Resource*>::iterator it = _pool.find(name);
    if (it == _pool.end())
        return NULL;
    else
        return it->second;
}

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

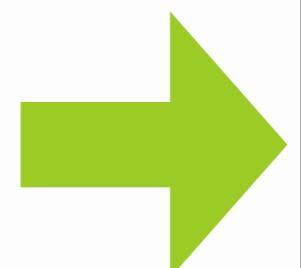
Resource::~Resource() { // destructeur
    unregisterResource(this); // auto-désenregistrement à la destruction
    cout << "Resource " << _name << " released and unregistered." << endl;
}

```

```
// fichier main.cc

int main(...)
{
    Resource *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.

182

Les exceptions

- Les exceptions forment un mécanisme d'interruption à l'exécution du programme
 - elles permettent une **gestion simplifiée et plus sûre des erreurs**
- Principe
 - une fonction qui détecte une erreur “**lance**” une exception (un objet) : **cela nécessite donc de détecter l'erreur et de lancer l'exception**, ce qui n'est pas automatique
 - **un gestionnaire d'exception** adapté **est consulté** (il “attrape” l'exception) **et un traitement approprié est exécuté**
 - lorsqu'une exception est lancée, **la pile d'appels de fonctions est remontée jusqu'à ce qu'elle soit rattrapée**

Les exceptions

> Try / catch

- Les exceptions sont “surveillées” dans un bloc particulier **try { }**
- Dans le bloc **try**, les exceptions sont “lancées” grâce à l’instruction **throw**
 - les exceptions sont typées : elles peuvent être d’un type de base (**int**, **double**...) ou un objet ou encore un pointeur sur un objet quelconque
 - ex : **long a; throw a;** ou même **throw Vector3(2.0);**
- Les exceptions sont “attrapées” et traitée dans un bloc particulier qui est protégé contre l’exception : **catch() {}**
 - un bloc **catch() {}** est écrit pour un type d’exception particulier

```

MyException e0("I/O Error"), e1("Memory allocation error");

try {      // début de zone sous surveillance
    switch(...) {
        case 0: {
            ...
            throw e0;
        } break;
        case 1: {
            ...
            throw e1;
        }
        case 2: {
            ...
            throw 0x01;
        }
        ...
    }
}

catch (MyException& me) { // après try
    cout << "Exception attrapée = " << me.name() << endl;
}
catch (int ie) { // intercepter les exceptions de type int
    cout << "Exception avec entier = " << ie << endl;
}
catch (...) { // les ... font partie de la syntaxe !
    // toutes les autres exceptions
    cout << "Oups, exception non prévue..." << endl;
}

```

on lance une exception particulière pour dérouter l'exécution vers le gestionnaire d'exception qui traitera le problème

les exceptions peuvent être directement lancées par `throw` ou par n'importe quelle fonction appelée à l'intérieur du bloc `try`

les exceptions sont des instances d'objets ou bien de types simples (entiers, réels etc.)

le (ou les) gestionnaire(s) d'exceptions apparaissent dans les blocs `catch()` {} après le bloc de surveillance `try{}`

Les exceptions

> Propagation d'une exception

- Une exception lancée “remonte” la pile d’appels des fonctions jusqu’à trouver un gestionnaire adapté

```
void func3(int ie) { throw ie; }
void func2(int ie) { func3(ie); }
void func(int ie) { if (!ie) func2(ie); else throw ie; }
```

```
int main() {
    try {
        func(0);
        func(3);
    }
    catch (int ie) {
        cout << "Exception levée avec id = " << ie << endl;
    }
    cout << "Fin" << endl;
    return 0;
}
```

cet appel n'est jamais effectué si une exception est lancée lors de l'exécution de l'appel précédent...

→
> Exception levée avec id = 0
> Fin

Les exceptions

> Relancer une exception

- Il est possible de relancer une exception (faire suivre), par exemple pour attraper une exception sans réellement effectuer le traitement nécessaire

```
try { ... }

catch (exception_type e) {
    throw f; // on relance une nouvelle exception à l'appelant
}

catch (exception_type e) {
    throw; // on relance l'exception e à l'appelant
}
```

Les exceptions

> Exception non prévue ?

- Si aucun gestionnaire adapté n'est trouvé pour une exception lors de la remontée de la pile de fonctions, la fonction `terminate()` est appelée : l'exécution du programme est interrompue

```
void func3(int ie) { throw 12.7; }
void func2(int ie) { func3(ie); }
void func(int ie) { if (!ie) func2(ie); else throw ie; }
```

```
int main() {
    try {
        func(0);
        func(3);
    }
    catch (int ie) {
        cout << "Exception levée avec id = " << ie << endl;
    }
    return 0;
}
```

on lance une exception de type `double` alors qu'on n'a prévu aucun gestionnaire adapté dans le programme

> `terminate` called after throwing
an instance of 'double'
> Abort trap

la fonction `terminate` est appelée quand aucun gestionnaire adapté n'est trouvé, elle quitte le programme avec la f° C standard `abort()`



Design patterns

- *Design pattern* = modèle de conception pour la résolution de problèmes récurrents
- Singleton
- Fabrique
- Foncteurs (objets fonctions) et templates
- Décorateurs
- Observateurs



Design pattern

> Singleton

- Problème : garantir l'existence d'une seule et unique instance d'une classe donnée
 - gestionnaire de ressources (son, graphique...), d'événements...
- Fournir un moyen de création/destruction du singleton
- Interdire la création/destruction d'autres instances
 - rendre constructeur et destructeur privés
- Fournir au programmeur un accès facile et direct au singleton
 - créer une méthode de classe dédiée
- Généralisation grâce aux templates

```

class ResourceManager {
public:
  static ResourceManager *getSingleton(* parametres *) {
    if (singleton == NULL) {
      singleton = new ResourceManager(* parametres *);
    }
    return singleton;
  }
private:
  ResourceManager(* parametres *) { /* initialisation du singleton */}
  ~ResourceManager() { /* destruction du singleton */}

  ResourceManager& operator= (const ResourceManager&) {return *this;}
  ResourceManager (const ResourceManager&) {}

  static ResourceManager* singleton;
};

```

une méthode de classe `getSingleton()` assure la création du singleton s'il n'est pas instancié et le retourne

```

ResourceManager* ResourceManager::singleton=NULL; // !! dans fichier .cc

int main (int argc, char * const argv[]) {
  ResourceManager m; //ERROR: 'ResourceManager::ResourceManager()' is private

  ResourceManager *manager = ResourceManager::getSingleton(); //OK

  delete manager; //ERROR: 'ResourceManager::~ResourceManager()' is private
  return 0;
}

```

on pourrait ajouter une méthode de classe `destroy()` pour libérer le singleton si nécessaire



Design pattern

> Fabrique

- Problème : mieux contrôler l'instanciation d'une classe
 - pour mieux gérer les erreurs (problème au moment de la création)
 - pour interdire les doublons (instances identiques)...
- Fournir un moyen de création d'une instance
 - méthode de classe `create()` : retourne l'instance créée
- Interdire la création directe d'instances (mais pas la destruction)
- Généralisation possible via les templates

```
class NamedResource {  
  
    string _name;  
  
public:  
    ~NamedResource() { _pool.erase(_name); }  
  
    static NamedResource *create(const string& name /* other parameters*/) {  
        map<string,NamedResource*>::iterator it;  
        if ((it=_pool.find(name)) != _pool.end())  
            return it->second;  
        else  
            return new NamedResource(name /*other parameters*/);  
    }  
  
private:  
    NamedResource(const string& name /*other parameters*/) : _name(name) {  
        _pool[_name] = this;  
    }  
  
    NamedResource& operator= (const NamedResource&) { return *this; }  
    NamedResource (const NamedResource&) {}  
  
    static map<string,NamedResource*> _pool;  
};
```

une méthode de classe `create()` assure la création d'une instance s'il elle n'existe pas déjà, l'instance existante de même nom est retournée sinon.

193

```
int main (int argc, char * const argv[]) {  
  
    NamedResource r("toto"); //Error: 'NamedResource::NamedResource(const  
std::string&)' is private  
  
    NamedResource *pr = NamedResource::create("toto");  
  
    delete pr; //OK (destructeur public)  
  
    return 0;  
}
```