LANGAGE OBJ. AV.
(C++)
MASTER 1

U.F.R. d’Informatique
Université de Paris Cité

RETOUR SUR LES CLASSES

La **déclaration** (dans Compte.hpp) du concept élémentaire de compte en banque pourrait être :

```
class Compte {
 int solde;
public:
 int getSolde();
 void deposer(int somme);
 void retirer(int somme);
};
```

Elle **suffit** à imaginer une utilisation :

```
#include "Compte.hpp"
...
Compte monCompte;
cout << “Reste “ << monCompte.getSolde() << endl;
monCompte.deposer(100);
monCompte.retirer(23);
```

La **définition** de la classe (dans le Compte.cpp)

```
void Compte::deposer(int s) { solde += s; }
void Compte::retirer(int s) { solde -= s; }
int Compte::getSolde() { return solde; }
```

se fait avec l’opérateur de portée ::

Attention, si vous oubliez la portée et écrivez dans Compte.cpp

```
void deposer(int s) { ...qq chose ... }
```

Vous définirez alors une fonction, ce qui peut être source d'une erreur bien cachée.
Ici, comme {...qq chose...} utilisera solde, le compilateur préviendra heureusement d'une erreur

Convention :

la **déclaration** d’une classe se fait dans un "fichier d’entête" MaClasse.hpp

la **définition** des méthodes de la classe se fait dans un fichier "MaClasse.cpp"

l’**utilisation** de la classe nécessite #include "MaClasse.hpp"

La **compilation** séparée se fait via le couple make/Makefile (voir cours 1)

le concept d'encapsulation c’est

des constituants agrégés

des moyens d'actions regroupés

=> un mécanisme permettant de cacher certains aspects

La protection se fait durant la déclaration.
On distingue des domaines
(private, public, protected, default)

```
class A {

      déclarations
    domaine:
      déclarations
    domaine:
      déclarations
};
```

Rq : un domaine peut apparaître plusieurs fois

Le domaine `private` :

ses éléments ne sont visibles que par des fonctions membres de la classe.

```
class A {
   private:
     int attribut;
     void methode1();
   public:
     void methode2();
};
```

```
void A::methode1() {
    attribut = 1; // ok
}
void A::methode2() {
    methode1();   // ok
    attribut = 2; // ok
}
int main() {
    A a;
    a.attribut;   // non
    a.methode1(); // non
    a.methode2(); // oui
}
```

Le domaine `public` :

visible, accessible partout

Le domaine `protected` :

est qq chose d'intermédiaire. C'est en gros une visibilité conservée par héritage

Le domaine par défaut :

c'est `private` (un peu différent de java donc)

Les **constructeurs** sont des méthodes qui :

portent le nom de la classe,
ne précisent pas de type retour
sont appelées lors de la création de l'objet

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
};
```

```
Compte::Compte(int x):solde{x} {  cout << "bienvenue" << endl; }
```

Notez bien la séquence d'initialisation avant le bloc

```
int main() {
 Compte c1{1000};
}
```

Les **constructeurs** sont des méthodes qui :

portent le nom de la classe,
ne précisent pas de type retour
sont appelées lors de la création de l'objet

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
 Compte();
};
```

```
Compte::Compte(int x):solde{x} {  cout << "bienvenue" << endl; }
Compte::Compte():Compte{0}{}
```

notez comment un constructeur fait appel à un autre

```
int main() {
 Compte c1{1000}, c2, c3{}; // mais pas faire c4() !
}
```

Le **concepteur** doit adopter une démarche centrée sur l'utilisateur :

Qu’est ce que l’utilisateur peut raisonnablement fournir comme données pour initialiser un objet ?

et non pas imposer que lui soit fournies exhaustivement toutes les données internes à l'objet...

Différence entre

```
Compte::Compte(int x):solde{x} {
    cout << "bienvenue" << endl;
 }
```

et

```
Compte::Compte(int x) {
    solde =x;
    cout << "bienvenue" << endl;
 }
```

où :

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
};
```

Ordre dans la création d'un objet :

- allocation de la mémoire (pour l'objet et ses champs)

- puis initialisation de toutes les données membres dans l’ordre de leur déclaration. Soit explicitement grâce à la séquence d'initialisation, soit par un constructeur par défaut.

- puis exécution du code du bloc du constructeur de la classe englobante

Mise en situation ...
sans séquence d'initialisation ???

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
class Segment{
    private:
      Point premier, second;
    public:
      Segment(int x1,int y1,
              int x2,int y2);
};
```

```
Point::Point(int x,int y) { abscisse = x; ordonnee = y; }
Segment::Segment(int x1,int y1,int x2,int y2){
  // les points premier et second sont censés être construits ...
  // mais comment ?
  // de plus leurs abs, ord seraient inaccessibles (private)
  // comment pourrait-on leur affecter x1,y1,x2,y2 ? ...
 }
```

Mise en situation ...
avec séquence d'initialisation
(les pbs disparaissent)

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
class Segment{
    private:
      Point premier, second;
    public:
      Segment(int x1,int y1,
              int x2,int y2);
};
```

```
Segment::Segment(int x1,int y1,int x2,int y2)
          : premier{x1,y1}, second{x2,y2} {}
```

Vous serez donc "critiquables" si vous écrivez :

```
Compte::Compte(int x) {
    solde =x;
 }
```

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
};
```

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point *p;
   return 0;
};
```

Ici aucun objet Point n'est construit

p manipule des adresses

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point *p{nullptr}, q{1,2};
   return 0;
};
```

p est initialisé à nullptr,

q est construit

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point q{1,2}, *p{&q};
   return 0;
};
```

q est construit

p est initialisé avec l'adresse de q,

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point q{1,2}, *p{&q};
   p=new Point(3,4);
   delete p;
   return 0;
};
```

new permet d'invoquer un constructeur
l'adresse de l'objet créé est retournée à p
*p est de type Point
(*p).methode() est autorisé
sémantiquement équivalent : p-> methode()
(syntaxiquement opérateurs différents ...)

Un objet peut parler de lui-même en utilisant le mot-clé `this`

son type est "pointeur vers le type de l'objet"

`this` permet :

de lever certaines *ambiguïtés*

de transmettre l'objet actif en argument d'une autre méthode ou de return

exemple de levée d’ambiguïté :

```
class A {
  private:
     int valeur;
    public:
      void setValeur(int valeur);
};
```

```
void A::setValeur(int valeur) {
   this->valeur = valeur;
}
```

exemple de besoin d'auto-nommage :

```
class A {
  private:
     int valeur;
    public:
     A min(A other);
};
```

```
A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return ???
}
```

exemple de besoin d'auto-nommage :

```
class A {
  private:
     int valeur;
    public:
     A min(A other);
};
```

```
A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return *this;  // pour respecter le type retour
}
```

Complément sur const ...

Déjà rencontré comme modificateur de type. L'usage des variables est alors restreint afin de "ne pas pouvoir les modifier".

const peut également être adossée à la spécification d'une méthode. La méthode s'engage alors à laisser l'objet qui l'exécute inchangé
(const s'applique alors aux attributs de l'objet : ce sont eux qui seront non modifiables).

```
class A {
private :
  int att;
public:
 A(int x);
 int get() const;
 void affiche() const;
 void set(int v);
};
```

Les méthodes get() et affiche() laisseront invariant l'objet qui les invoquent

La contrainte const peut être jugée trop forte...

On veut pouvoir modifier certains attributs techniques alors même que du point de vue d'un observateur extérieur il sera tout de même considéré comme constant...

Exemple : un compte en banque dont on imagine que les transactions sont toutes tracées y compris les consultations...
Un utilisateur peut considèrer qu'après un get() le compte est grosso-modo inchangé

```
class Compte {
private:
 int solde;
 int nbTransactions;
public:
 int getSolde() const {
    nbTransactions++;    // interdit par le const
    return solde;
 }
};
```

```
class Compte {
private:
 int solde;
 mutable int nbTransactions;
public:
 int getSolde() const {
    nbTransactions++;    // autorisé malgré tout !
    return solde;
 }
};
```

On a vu dans cette première partie essentiellement des choses que vous connaissiez :

- définition des classes (private etc ..)
- construction (à la c++)
- pointeurs et objets :

x->m() ou (*x).m() ; new + ctor ; delete

- this
- const pour les méthodes / objets
- exception à const : mutable
- place aux nouveautés maintenant :

    destructions, autres copies, références

Destruction

Durée de vie des objets

C++ offre un moyen de réaliser des opérations lorsqu’un objet disparaît : elles sont à préciser dans un **destructeur**

Chaque classe en possède.

C'est l'alter-ego des constructeurs...

Un **destructeur** est :

- Une méthode dont le nom est celui de la classe préfixée par le caractère ~
- ne déclarant rien à renvoyer, ne prend pas de paramètre
- un seul destructeur par classe
- il faut le qualifier virtual (on verra plus tard pourquoi)
- il est appelé automatiquement à la destruction d'un objet.

Une destruction est :

- implicite dans le cas d'un objet déclaré lorsque le point de contrôle quitte le bloc de cette déclaration
- explicite (via delete) lorsqu'on décide de détruire un objet alloué dynamiquement (un pointeur obtenu par new) il est appelé avant de libérer la mémoire

Exemple 1 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void f(){
    cout << "dans f" << endl;
    A x('x');
}
int main() {
    A a('a'),b('b');
    f();
    f();
    return 0;
}
```

```
Naissance de a
Naissance de b
dans f
Naissance de x
Mort de x
dans f
Naissance de x
Mort de x
Mort de b
Mort de a
```

Rq : autant de mort que de naissance !

Exemple 2 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
dans g
Mort de a
Mort de a
```

ici on dirait que "a" meurt 2 fois ...

Exemple 2 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
dans g
Mort de a
Mort de a
```

on comprend que c'est la trace de la mort de y, ouf ... c'est plus clair ...

Exemple 2 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
dans g
Mort de a
Mort de a
```

ouf ?? il manque qd même des naissances !

Explication :
visiblement la déclaration/initialisation des paramètres au moment de l'appel de fonction ne fait pas appel au constructeur que nous avons écrit (sinon on aurait une "naissance")

On comprend, en notant la mort du paramètre, que c++ a opéré une sorte de clonage en recopiant argument pour argument l'objet transmis.

Ce mécanisme est appelé construction par copie (d'un original).

Il existe par défaut, mais on peut aussi le redéfinir explicitement (et donc témoigner d'une "naissance")

Présentation du constructeur de copie

```
class A {
public:
 char name;
 A(char);
 A (const A&)
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;}
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de "<< name << endl;}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a
Mort de a
```

ici tout redevient cohérent

Focus sur le constructeur de copie

```
class A {
public:
 A (const A&)
...
};
```

```
...
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de
"<< name << endl;}
```

remarques :

- à la création on se charge, dans la séquence d'initialisation de faire se correspondre les attributs (on pourrait donc faire autrement !?)
- on a déclaré le constructeur de copie public (on pourrait donc faire autrement !?)
- la copie déclare laisser son argument constant (on pourrait donc faire autrement !?)
- le symbole & est utilisé dans l'argument (à justifier !?)

à la création on se charge, dans la séquence d'initialisation de faire se correspondre les attributs (on pourrait donc faire autrement !)

```
class A {
public:
 string name;
 A(char);
 A (const A&)
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;}
A::A (const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name << endl;}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a_copie
Mort de a
```

on a déclaré le constructeur de copie public
(on pourrait donc faire autrement !)
c.à d interdire de passer un objet en argument !

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
private :
 A (const A&)
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;}
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de "<< name << endl;}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
main.cpp: error:
‘A::A(const A&)’ is private
within this context
     g(a);
         ^
```

la copie déclare laisser son argument constant (on pourrait donc faire autrement !)

```
class A {
public:
 string name;
 A(char);
 A ( A&)
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;}
A::A ( A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name << endl;
  x.name='*';
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a_copie
Mort de *
```

Attention donc au constructeur de copie !

A retenir :

- Le constructeur de copie est implicitement impliqué dans la transmission des objets en arguments
- Il a une définition par défaut qui copie membre à membre les attributs
- Sa signature habituelle pour une classe A est

```
public:
 A (const A&);
```

- Il est possible de le redéfinir largement.

(En conséquence la "copie" peut ne pas en être une ! Son utilisation "intuitive" peut s'en trouver totalement perturbante)

Il nous reste à parler de ce & dans

```
public:
 A (const A&);
```

Nous avons rencontré le symbole & au moment où nous avons parlé des pointeurs.

```
int a{3}, *pa{&a};
```

Il s'agissait d'obtenir une valeur : l'adresse d'une variable

L'usage de & ici est totalement différent :

nous allons parler de **transmission par référence**

(Nous allons d'abord justifier de son utilité)

Rappel :
À l’entrée d’une fonction recevant un argument, on créé une nouvelle variable du type considéré. Son nom est celui du paramètre formel. Elle est initialisée à l’aide de la valeur du paramètre effectif (et nous venons de voir que cette initialisation est faite par le constructeur de copie)

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a('a');
    g(a);
    return 0;
}
```

Si on imaginait une signature pour le constructeur de copie de la forme

```
class A {
public:
 A(A original); //incorrect
};
```

Sachant que le constructeur par copie est invoqué à chaque passage d'argument, et qu'il a lui même un argument : il y aurait un pb récursif : à l'entrée du constructeur, "original" devrait être initialisé par copie etc ...

```
class A {
public:
 A(A &original); //correct
};
```

On comprend qu'on a besoin d'un autre mode de passage de paramètre, sans copie.

Le symbole & indique qu'on ne créé pas de nouvel objet pour cette variable.
Il exprime simplement qu'on désigne le paramètre transmis à l’aide d’un nom, comme un alias qui sera local.

L'argument transmis et cet alias désignent donc le même objet.

La transmission par référence,
ce qu'il faut retenir pour l'instant :

Elle est désignée par une nouvelle utilisation de &,
(différente des opérations sur les pointeurs)

Nous l'avons rencontrée dans la signature du constructeur par copie :

```
class A {
public:
 A(A & original);
};
```

Elle s'interprète comme un mécanisme de création d'un alias du paramètre, il permet de désigner strictement le même objet, celui transmis et celui identifié localement.

La transmission par référence,
ce qu'il faut retenir pour l'instant :

Elle est désignée par une nouvelle utilisation de &,
(différente des opérations sur les pointeurs)

Nous l'avons rencontrée dans la signature du constructeur par copie :

```
class A {
public:
 A(A & original);
};
```

Elle s'interprète comme un mécanisme de création d'un alias du paramètre, il permet de désigner strictement le même objet, celui transmis et celui identifié localement.

Il reste des chose à approfondir, mais nous n'en avons pas encore terminé avec la copie.

Que se passe t'il lors du retour d'une fonction ?

```
class A {
public:
 string name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
A h() {
 cout << "dans h" << endl;
 A z('z');
 return z;
}

int main() {
    cout << "un A " << h().name <<
endl;

    return 0;
}
```

Que se passe t'il lors du retour d'une fonction ?

```
class A {
public:
 string name;
 A(char);
 A ( const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
A h() {
 cout << "dans h" << endl;
 A z('z');
 return z;
}

int main() {
    cout << "un A " << h().name <<
endl;

    return 0;
}
```

```
un A dans h
Naissance de z
z
Mort de z
```

curieux : le cycle de la vie semble non naturel

Que se passe t'il lors du retour d'une fonction ?

```
class A {
public:
 string name;
 A(char);
 A ( const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
A h() {
 cout << "dans h" << endl;
 A z('z');
 return z;
}

int main() {
    cout << "un A " << h().name <<
endl;

    return 0;
}
```

```
un A dans h
Naissance de z
z
Mort de z
```

le retour "devrait" produire une copie (anonyme)

Attention aux optimisations du compilateur

"RVO Return Value Optimization"

car dans certains cas le compilateur essaie de minimiser la construction d’objets...

Pour supprimer cette optimisation il faut utiliser (avec gcc) l’option `-fno-elide-constructors`

Avec l'option -fno-elide-constructors :

```
class A {
public:
 string name;
 A(char);
 A ( const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name << endl;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name << endl;
}
A::~A() { cout << "Mort de " << name << endl;}
```

```
A h() {
 cout << "dans h" << endl;
 A z('z');
 return z;
}

int main() {
    cout << "un A " << h().name <<
endl;

    return 0;
}
```

```
un A dans h
Naissance de z
Naissance d'une copie de z
Mort de z
z_copie
Mort de z_copie
```

plus cohérent ! Attention donc ...

Sans transition ...

L'affichage traditionnel en c++ se fait avec cout.

Comment l'étendre à d'autres objets ?

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

<< est en fait un opérateur binaire. Il prend :

- un argument de type ostream
- un second argument (soit int, soit string, ...)

Comme toute fonction on peut le surcharger
(l'intérêt est de le faire sur son second argument)

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

<< est en fait un opérateur binaire. Il prend :

- un argument de type ostream
- un second argument (int, string, ...)

Comme toute fonction on peut le surcharger
(l'intérêt est de le faire sur son second argument)

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

```
ostream & operator<<( ostream &out , const int &x );
ostream & operator<<( ostream &out , const string &x );
```

ce sont ces deux redéfinitions, qui existent déjà, qui les rendent possible

On procède ainsi (on expliquera ensuite)

Dans A.hpp :

```
#include <iostream>
using namespace std;
class A {
public:
 string name;
 etc ...
};
ostream& operator<<( ostream &out , const A &x );
```

Notez que la déclaration se fait à l'extérieur de celle de la classe : ce n'est pas une fonction membre. Mais il est naturel de la placer dans A.hpp

Puis, dans A.cpp :

```
ostream& operator<<(ostream &out ,const A&x ) {
  out << x.name ;
  return out ;
}
```

Notez encore que l'on ne préfixe pas par A::
(toujours parce que ce n'est pas une fonction membre)

décryptage :
  - lors de l'appel c'est la constante cout qui sera transmise au 1er paramètre de l'opérateur : out
  - cette transmission se fera par un alias (notez le passage par référence &) et pas par copie, par économie
  - le second argument est de type A
  - lui aussi est transmis tel quel, sans copie
  - on précise même ici qu'il ne sera pas modifié (const) car seule une lecture est utile. (On peut choisir que non)
  - dans le bloc on utilise une autre surcharge de operator<< celle dont le second argument est une string
  - que le type retour soit l'ostream d'entrée permet, par associativité d'enchaîner cout << truc << bidule;

```
ostream& operator<<(ostream &out ,const A&x ) {
  out << x.name ;
  return out ;
}
```