# Initialisation, conversion, surchage et redéfinition variables et méthodes de classes

Objets avancés, HLIN603

5 février 2015

# Sommaire

# Cas d'étude : la classe Date

```
//------------------ Date.h ----------------------------
#ifndef Date_h
#define Date_h
class Date
{
private:
     int annee, mois, jour;
public:
     Date();
     Date(int,int,int);
     virtual ~Date();
     virtual int getAnnee()const;
     virtual int getMois()const;
     virtual int getJour()const;
     virtual void setAnnee(int);
     virtual void setMois(int);
     virtual void setJour(int);
     virtual void affiche(ostream&)const;
     virtual bool operator<(Date d)const;
     // volontairement passage par copie
};
```

# Cas d'étude : la classe Date

```
//------------------ Date.cc ----------------------------
#include <iostream> #include <string> using namespace std;
#include "Date.h"
Date::Date(){annee=2000; mois=12; jour=31;}
Date::Date(int a,int m,int j){annee=a; mois=m; jour=j;}
Date::~Date(){}
int Date::getAnnee()const{return annee;}
int Date::getMois()const{return mois;}
int Date::getJour()const{return jour;}
void Date::setAnnee(int a){annee=a;}
void Date::setMois(int m){mois=m;}
void Date::setJour(int j){jour=j;}
void Date::affiche(ostream&)const
{cout << jour << "/" << mois << "/" << annee << endl;}
bool Date::operator<(Date d)const{
return
( (this->annee < d.annee) ||
(this->annee == d.annee && this->mois < d.mois) ||
(this->annee == d.annee && this->mois == d.mois && this->jour<d.jour));
}
```

# Initialisation versus affectation

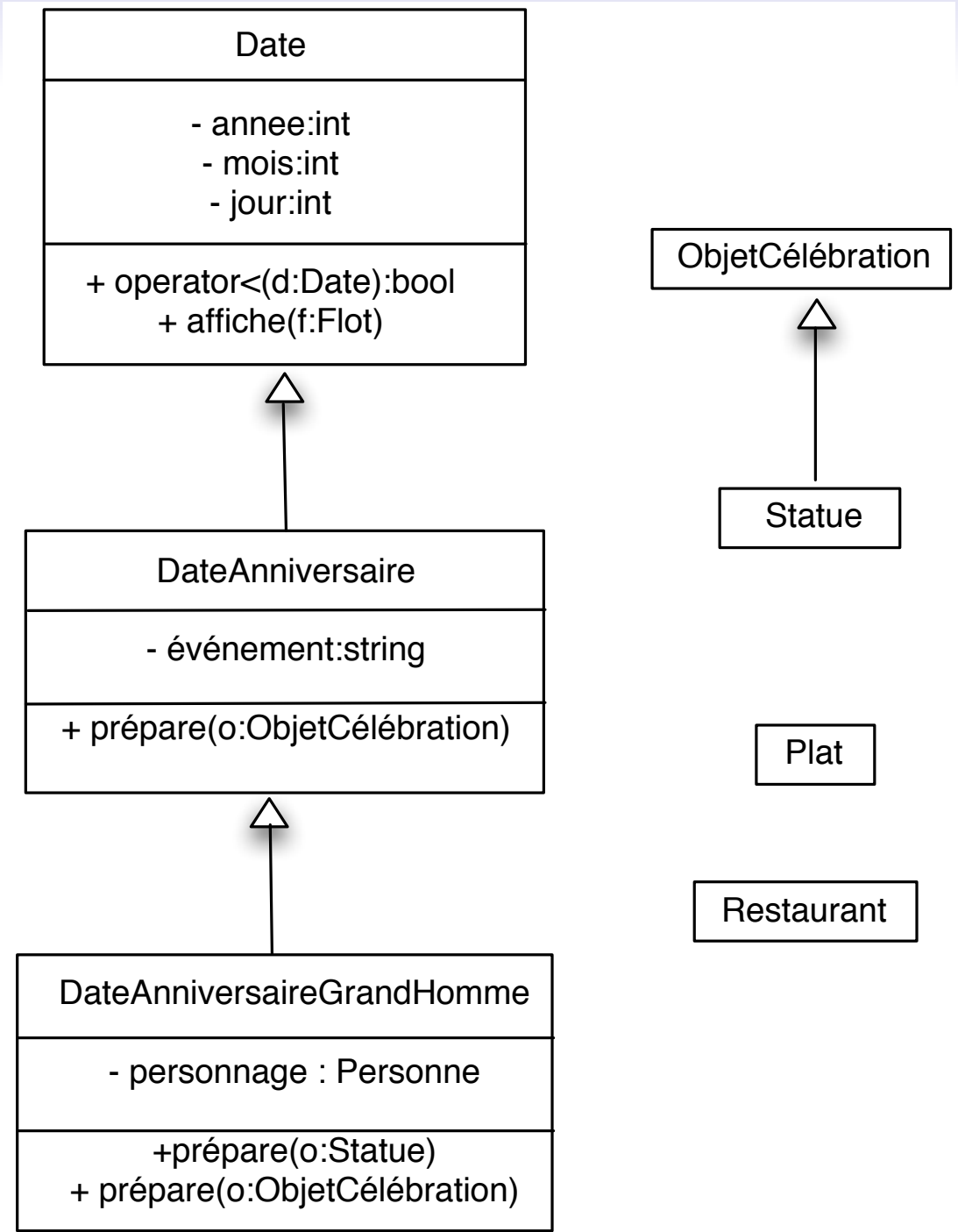
Le symbole = ne fait pas toujours référence à l'opérateur d'affectation :

```
Date d1;
Date d2=d1;
```

équivaut à `Date d1, d2(d1);`

n'équivaut pas à : `Date d1, d2; d2=d1;`

Encore une raison de garder en cohérence *operator* = et constructeur par copie pour que les deux expressions fonctionnent de la même manière


Date
- annee:int
- mois:int
- jour:int
+ operator<(d:Date):bool
+ affiche(f:Flot)
ObjetCélébration
Statue
DateAnniversaire
- événement:string
+ prépare(o:ObjetCélébration)
Plat
Restaurant
DateAnniversaireGrandHomme
- personnage : Personne
+prépare(o:Statue)
+ prépare(o:ObjetCélébration)

# Rappel : conversion standard d'un type (classe) vers un super-type (super-classe)

```
DateAnniversaire *da = new DateAnniversaire();
DateAnniversaire *dagh = new DateAnniversaireGrandHomme();
DateAnniversaireGrandHomme *dagh2 = new DateAnniversaireGrandHomme();

//conversions standards sous-type vers super-type
da=dagh2; // oui : affectation polymorphe
//dagh2=dagh; non ! il faut effectuer une coercition explicite
dagh2=dynamic_cast<DateAnniversaireGrandHomme *>(dagh);

//affectation
*da=*dagh2; //ok mais copie seulement la partie DateAnniversaire
// *dagh2=*da; non sauf si on definit un operateur =
// dans DateAnniversaireGrandHomme
// avec comme parametre une DateAnniversaire
}
```

# Conversion basée sur un constructeur

Règle : un constructeur avec un unique paramètre est considéré comme un opérateur de conversion

```
//----- Dans Date.h --------
class Date{...
Date(string s);
...
};

//----- Dans Date.cc --------
Date::Date(string s) // s de format 'yyyymmdd'
{
  annee=1000*(s.at(0)-'0')+100*(s.at(1)-'0')
              +10*(s.at(2)-'0')+(s.at(3)-'0');
  mois=10*(s.at(4)-'0')+(s.at(5)-'0');
  jour=10*(s.at(6)-'0')+(s.at(7)-'0');
}
```

# Conversion basée sur un constructeur

Usage explicite du constructeur de `Date` à partir d'une `String`

```
//utilisation explicite du constructeur comme opérateur de conversion

// création d'un objet, allocation automatique
Date stSylvestre("20041231");
stSylvestre.affiche(cout);

// création d'un objet volatile
Date *d2 = new Date("20120209");
cout << (*d2 < Date("20041231")) << endl;
```

# Conversion basée sur un constructeur

Usage implicite du constructeur de `Date` à partir d'une `String`

```
//utilisation implicite du constructeur comme opérateur de conversion
string s="20041231";
cout << (*d2 < s) << endl;
```

Ce peut être :

- surprenant
- dangereux !

# Conversion basée sur un constructeur

Interdire l'usage implicite du constructeur de `Date` à partir d'une `String`
mot-clef `explicit`

```
class Date
{
.....
public:
....
explicit  Date(string s);
....
};
```

# Opérateurs de conversion

Le procédé basé sur les constructeurs n'est pas toujours applicable :

- lorsque l'on veut convertir vers un type de base (qui n'est pas une classe et ne peut avoir de constructeur),
- lorsque l'on veut convertir vers une classe qui est déjà définie et que l'on ne veut pas ou ne peut pas modifier.

Dans ces cas, on définit un opérateur de conversion

# Opérateur de conversion de Date vers string

```
//----- Dans Date.h --------
class Date{...
public:
....
//opérateur de conversion de Date vers string
virtual operator string()const;
...
};
```

# Opérateur de conversion de Date vers string

```
Date::operator string()const   //----- Dans Date.cc --------
{    string s="";   int reste;
int chiffreMilleAnnee=annee/1000; reste=annee%1000;
int chiffreCentAnnee=reste/100; reste=reste%100;
int chiffreDizAnnee=reste/10; reste=reste%10;
int chiffreUnitAnnee=reste;
s+= (char)(chiffreMilleAnnee+((int)'0'));
s+=(char)(chiffreCentAnnee+((int)'0'));
s+=(char)(chiffreDizAnnee+((int)'0'));
s+=(char)(chiffreUnitAnnee+((int)'0'));

int chiffreDizMois=mois/10; reste=mois%10; int chiffreUnitMois=reste;
s+=(char)(chiffreDizMois+((int)'0'));
s+=(char)(chiffreUnitMois+((int)'0'));

int chiffreDizJour=jour/10; reste=jour%10; int chiffreUnitJour=reste;
s+=(char)(chiffreDizJour+((int)'0'));
s+=(char)(chiffreUnitJour+((int)'0'));
return s;
}
```

# Opérateur de conversion de Date vers string

```
// ajout d'une nouvelle fonction
void f(string s){cout << "f" << s << endl;}

int main()
{
Date stSylvestre2004("20041231");
Date *halloween2004=new Date(2004,10,31);

// utilisation explicite de l'opérateur de conversion Date vers string

cout << (string(*halloween2004)) << endl;

// utilisation implicite de l'opérateur de conversion Date vers string

// avec l'opérateur d'affectation de la classe string
string s2; s2=stSylvestre2004;

//avec l'appel de f qui attend une string
f(stSylvestre2004);
```

# Ambiguïtés

Cas d'étude avec conversions basées sur des constructeurs :

- Date $\rightarrow$ Personne
- Date $\rightarrow$ Voiture

```
class  Personne
{
private:
Date anneeNaissance;
public:
Personne(Date d){anneeNaissance=d;}
};

class Voiture
{
private:
Date anneeConstruction;
public :
Voiture(Date d){anneeConstruction=d;}
};
```

# Ambiguïtés

Deux fonctions attendant resp. Personne ou voiture

```
void ecritAnnee(Personne p){cout << "ecrit annee personne"
<< p.getAnneeNaissance() << endl;}

void ecritAnnee(Voiture p){cout << "ecrit annee voiture"
<< p. getAnneeConstruction() << endl;}
```

Un malheureux programme

```
ecritAnnee(stSylvestre);
```

$\rightarrow$ message d'erreur signalant une ambiguïté.
Solution : apparier exactement la signature avec la forme d'appel

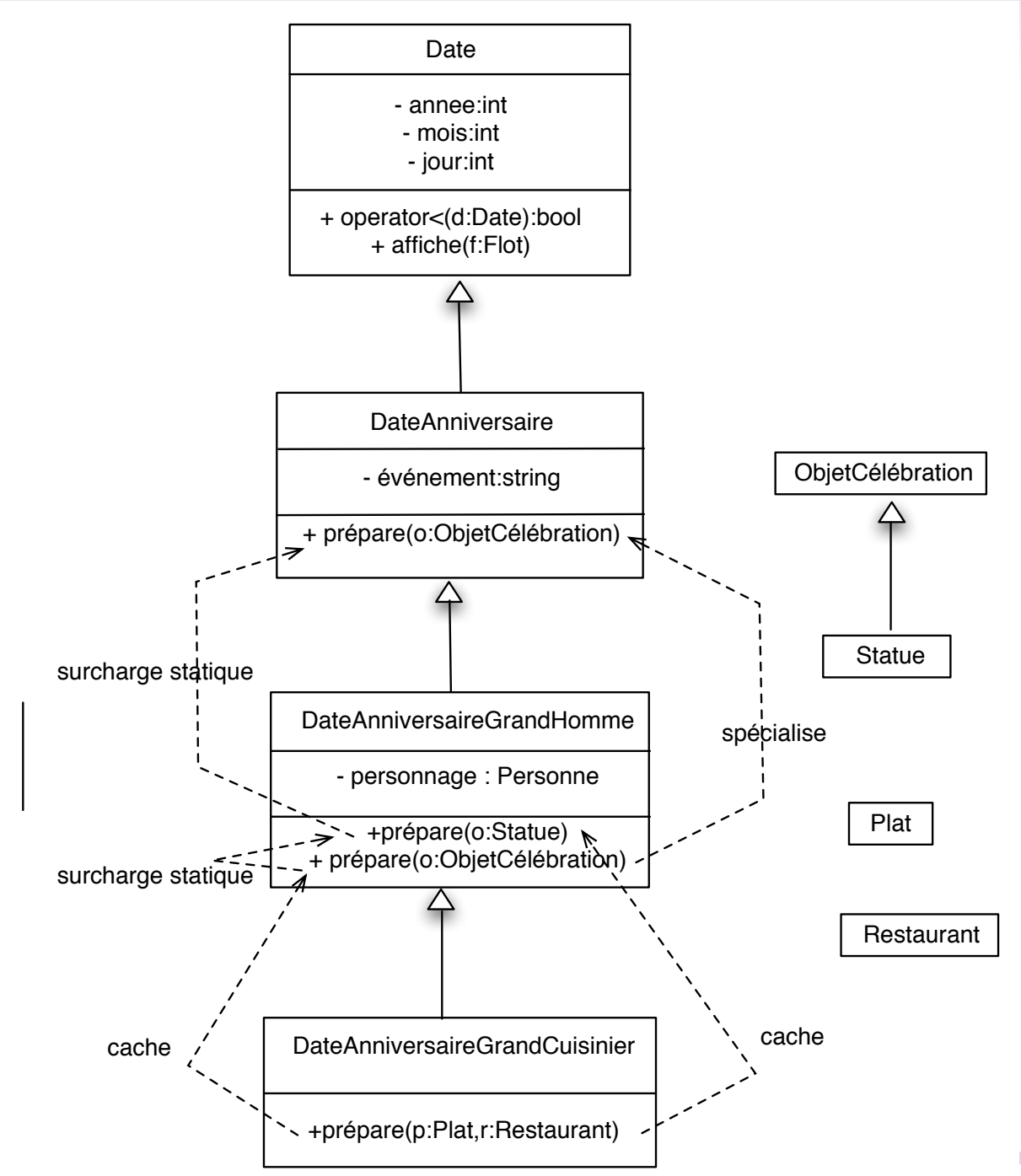
```
void ecritAnnee(Date d){cout << "ecrit annee Date" << endl;}
```

# Règle et moralité

Priorité avec laquelle le compilateur effectue ses choix :

1. s'il y a un appariement exact, il est utilisé. Des conversions très simples peuvent être effectuées : tableau vers pointeur, nom de fonction vers pointeur de fonction ou type `T` vers `const T`.
2. conversion vers un type englobant (`bool` vers `int`, `short` vers `int`, `float` vers `double`, etc.)
3. conversion standard : `int` vers `double` ou l'inverse, `T*` vers `void*`, pointeur vers une classe se transforme en pointeur vers une super-classe (affectation polymorphe)
4. utiliser les conversions définies par l'utilisateur (constructeurs et opérateurs de conversion)
5. utiliser l'ellipse (trois points `...` qui servent à dire que la liste des paramètres n'est pas connue) dans une déclaration de fonction

# Sommaire


Date
- annee:int
- mois:int
- jour:int
+ operator<(d:Date):bool
+ affiche(f:Flot)
DateAnniversaire
- événement:string
+ prépare(o:ObjetCélébration)
ObjetCélébration
Statue
surcharge statique
DateAnniversaireGrandHomme
- personnage : Personne
+prépare(o:Statue)
+ prépare(o:ObjetCélébration)
spécialise
Plat
surcharge statique
Restaurant
cache
DateAnniversaireGrandCuisinier
+prépare(p:Plat,r:Restaurant)
cache

# Précisions sur le code

```
class DateAnniversaire : virtual public Date{
public:
virtual void prepare(ObjetCelebration& o)
          {cout << "da-prepare-objet" << endl;}
};

class DateAnniversaireGrandHomme : virtual public DateAnniversaire{
public:
virtual void prepare(Statue&){cout << "dagh-prepare-statue" << endl;}
virtual void prepare(ObjetCelebration&)
         {cout << "dagh-prepare-objet" << endl;}
};

class DateAnniversaireGrandCuisinier :
          virtual public DateAnniversaireGrandHomme{
public:
virtual void prepare(Plat&,Restaurant&)
          {cout << "dagh-prepare-plat-restau" << endl;}
};
```

# Tout fonctionne bien

Compilation : examen du type statique (le type de la variable) pour choisir une signature compatible dans la classe.

Exécution : utilisation du type de l'objet (déterminé par le `new`) pour choisir la méthode ayant cette signature la plus spécialisée pour l'objet.

```
Restaurant r; Plat p; ObjetCelebration coupe; Statue s;

//----- appel de la méthode racine - choix de prepare(ObjetCelebration)

DateAnniversaire *da = new DateAnniversaire();
da->prepare(coupe); // da-prepare-objet
```

# Tout fonctionne bien

Compilation : examen du type statique (le type de la variable) pour choisir une signature compatible dans la classe.

Exécution : utilisation du type de l'objet (déterminé par le `new`) pour choisir la méthode ayant cette signature la plus spécialisée pour l'objet.

```
DateAnniversaire *dagh = new DateAnniversaireGrandHomme();
//----- appel de la méthode spécialisée -
// choix de prepare(ObjetCelebration)
dagh->prepare(coupe); // dagh-prepare-objet
//----- appel de la méthode spécialisée -

// choix de prepare(ObjetCelebration)
// car prepare(Statue) est une signature qui n'existe pas
// dans DateAnniversaire
dagh->prepare(s); // dagh-prepare-objet
```

# Tout fonctionne bien

Compilation : examen du type statique (le type de la variable) pour choisir une signature compatible dans la classe.

Exécution : utilisation du type de l'objet (déterminé par le `new`) pour choisir la méthode ayant cette signature la plus spécialisée pour l'objet.

```
DateAnniversaireGrandHomme *dagh2 = new DateAnniversaireGrandHomme();
//----- appel de la méthode spécialisée -
// choix de prepare(ObjetCelebration)
dagh2->prepare(coupe); // dagh-prepare-objet

//----- appel de la méthode surchargée - choix de prepare(Statue)
dagh2->prepare(s); // dagh-prepare-statue
```

# Rien ne va plus

```
DateAnniversaireGrandCuisinier *dagc
                        = new DateAnniversaireGrandCuisinier();
//----- appel de la méthode surchargée
dagc->prepare(p,r); // dagh-prepare-plat-restau

//essai d'appel des méthodes héritées -- > erreurs de compilation !!!!
//dagc->prepare(coupe);
//dagc->prepare(s);
}
```

- `prepare(Plat, Restaurant)` devrait simplement surcharger, c'est-à-dire cohabiter avec, les autres méthodes (héritées) `prepare`
- en réalité elle les cache!!!!!

# Une drôle de solution

... peu satisfaisante !

```
class DateAnniversaireGrandCuisinier :
              virtual public DateAnniversaireGrandHomme{
public:
virtual void prepare(Plat&,Restaurant&){.....}

// on les ajoute si on veut continuer a en disposer

virtual void prepare(ObjetCelebration&)
          {cout << "dagh-prepare-objet" << endl;}
virtual void prepare(Statue&)
         {cout << "dagh-prepare-statue" << endl;}
};
```

# Sommaire

# Définition

Les attributs et les méthodes dites « de classe » expriment, selon les cas, des données et des opérations :

- relatives à la classe toute entière et non à une instance particulière,
- partagées par toutes les instances de la classe (ce qui comprend les instances des sous-classes).

# Exemples d'attributs/variable de classe

*classe Carré*
*variable de classe NombreCôtés, constante*

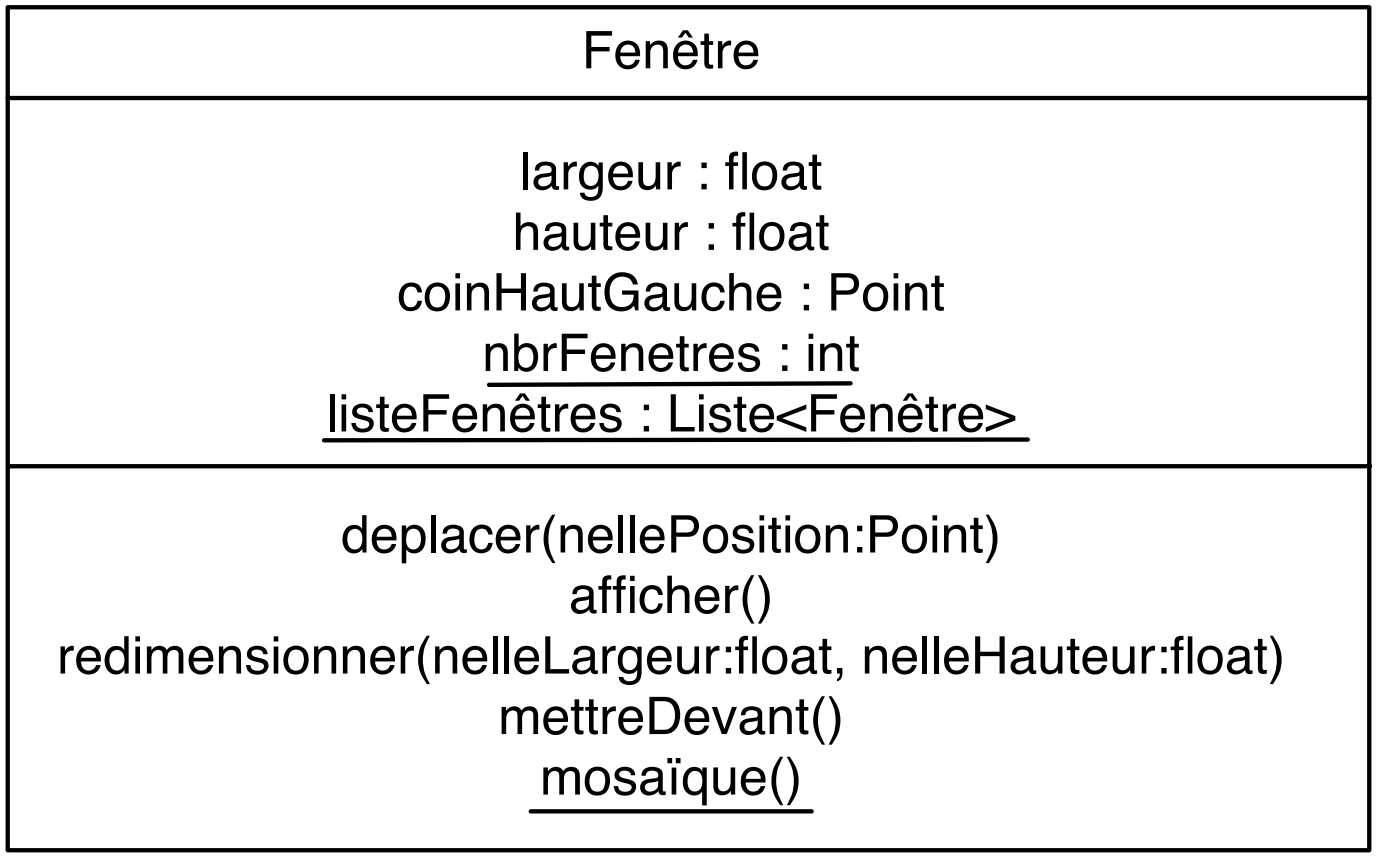
*classe Français*
*variable de classe PrésidentDeLaRépublique*

*classe Date*
*variable de classe Mois, contient la collection des noms des mois de l'année*

# Exemples de méthodes de classe

- Affichage des variables de classe
- Création d'une instance prototypique de la classe
- Calculs sur l'ensemble des instances (statistiques ...)
  plus généralement gestion de l'ensemble des instances
  ... Mais on peut aussi écrire une classe "gestionnaire", c'est même conseillé !

# Une classe `Fenetre` en UML



# Analyse de la classe `Fenetre`

| | | |
|---|---|---|
| **attributs d'instance** | Largeur, Hauteur : | flottant |
| | CoinHautGauche : | Point |
| **attributs de classe** | NbrFenêtres : | entier |
| | ListeFenêtres : | Liste<Fenêtre> |
| **méthodes d'instance** | Deplacer(...) | |
| | Afficher(...) | |
| | Redimensionner(...) | |
| | MettreDevant(...) | |
| **méthodes de classe** | Mosaïque() | |

# Une classe Fenetre en C++

```
//------------- Fenetre.h ----------------------
class Fenetre
{
private :
// attributs d'instance
float Largeur, Hauteur;
Point CoinHautGauche;

// Declaration des attributs de classe
static vector<Fenetre> ListeFenetre;
public :
static int NbrFenetres;
// exceptionnellement public pour montrer l'utilisation
```

# Une classe Fenetre en C++

```
//------------- Fenetre.h ----------------------
class Fenetre
{
.......

// Methodes d'instance
virtual void Deplacer(Point&);
virtual void Afficher(...);
virtual void Redimensionner(...);
virtual void MettreDevant(...);

// Methodes de classe
static void Mosaique();

// Miscellaneous
Fenetre();
virtual ~Fenetre();
};
```

# Une classe Fenetre en C++

Le fichier d'implémentation (d'extension .cc) doit contenir les définitions des attributs de classe, même si on ne les initialise pas.

```
//------------- Fenetre.cc ----------------------

// Definition et initialisation eventuelle
int Fenetre::NbrFenetres=0;
vector<Fenetre> Fenetre::ListeFenetre;

void Fenetre::Mosaique() { .... }
```

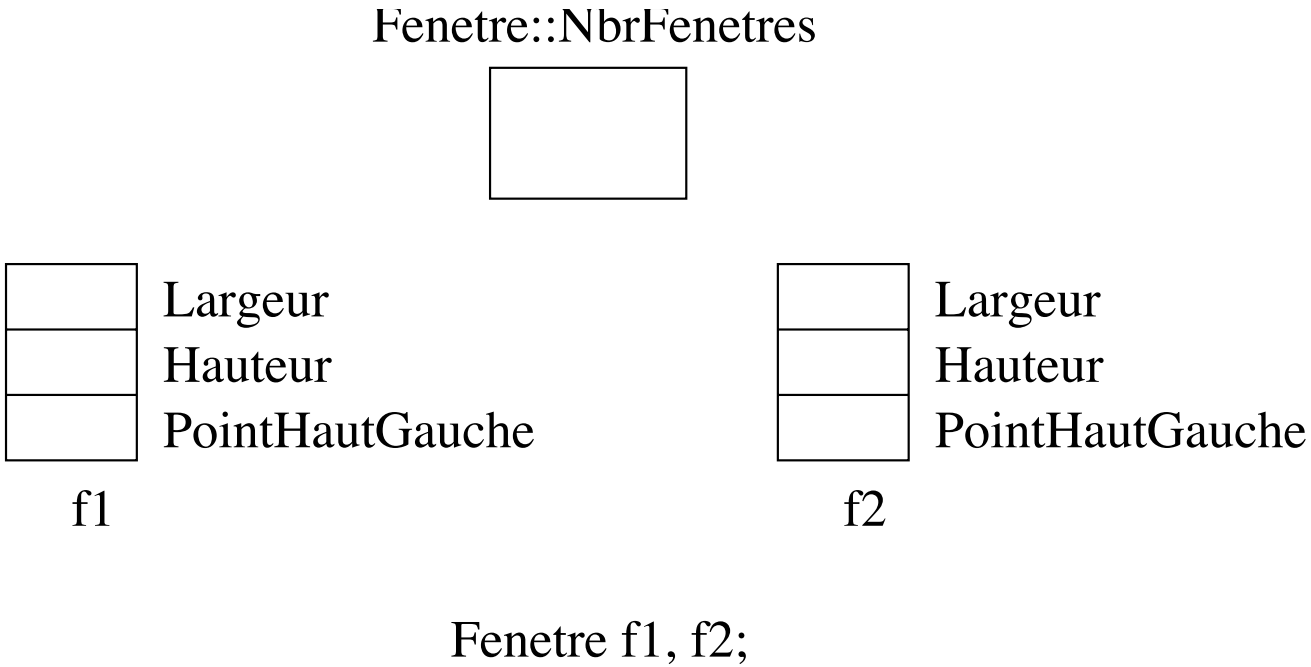
# Une classe Fenetre en C++

En dehors de la portée de la classe (par exemple dans un main), on utilise ces variables et ces méthodes, lorsqu'elles sont accessibles, en les préfixant par le nom de la classe suivi de l'opérateur de portée (::).

```
//-------------Main Fenetre.cc ----------------------

int main()
{
  Fenetre::Mosaique();
  cout << Fenetre::NbrFenetres;
}
```

# Une classe `Fenetre` en C++




- A chaque fenêtre ses attributs d'instances
- Un exemplaire des attributs de classes

# Ne pas abuser des attributs et méthodes de classe

- révèlent une conception souvent inaboutie : introduire une classe `Gestionnaire de fenêtres` qui recevrait toutes les caractéristiques statiques de la classe fenêtre serait mieux !
- méthodes ni héritées, ni liées dynamiquement, `this` n'a pas de sens à l'intérieur.