Programmation Orientée Objets 2
L3 MI
Cours 2

Rémi Watrigant[1]

[1] basé sur le cours de Marianne Huchard

Sommaire du cours 2

Sommaire

Classification et approches à objets

Rapprocher monde réel et représentation informatique
Rappel : un concept (ou classe) :

- *extension* : ensemble des objets couverts par le concept
- *intension* : ensemble des prédicats/propriétés vérifiés par les objets couverts

Exemple

Le concept de ***Rectangle*** *:*

- *extension : l'ensemble des rectangles*
- *intension : posséder quatre côtés parallèles deux à deux, posséder deux côtés consécutifs formant un angle droit... etc.*

Classification et approches à objets

Rapprocher monde réel et représentation informatique
Spécialisation/héritage :

- inclusion des *extensions*
- raffinement des *intensions*

Exemple

Carré** spécialise **Rectangle

- *L'ensemble des carrés est contenu dans l'ensemble des rectangles*
- *un carré a toutes les propriétés d'un rectangle + tous les côtés de longueur égale*

Classification et approches à objets

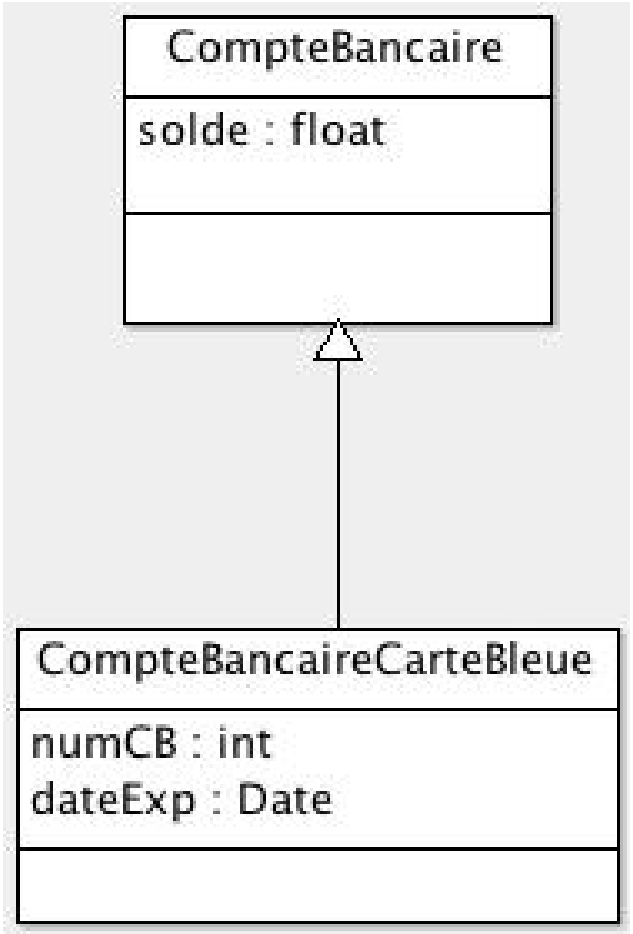
Formes de la classification :

- Héritage simple $\Rightarrow$ arborescence (Smalltalk, classes Java...)
- Héritage multiple $\Rightarrow$ graphe sans circuits (C++, Eiffel, interfaces Java...)

Dans ce cours : héritage simple en C++

Sommaire

Cas d'étude


CompteBancaire
solde : float
CompteBancaireCarteBleue
numCB : int
dateExp : Date

Déclaration dans les fichiers *header*

```
// Compte bancaire
class CompteBancaire
{
private:
  float solde;
 public:
  CompteBancaire();
  virtual ~CompteBancaire();
};

// Compte bancaire avec carte bleue
class CompteBancaireCarteBleue : public virtual  CompteBancaire
{
private:
  int numCB; Date dateExp;
 public:
  CompteBancaireCarteBleue();
  virtual ~CompteBancaireCarteBleue();
};
```

Sommaire

Ordres d'appel

Constructeurs :

- du haut vers le bas :
  des super-classes vers les sous-classes
- attributs dont le type est une classe (pas un pointeur) :
  constructeur appelé juste avant le constructeur de la classe qui déclare l'attribut

Destructeurs

- sens inverse de la construction

Ordres d'appel

```
CompteBancaireCarteBleue * compte = new CompteBancaireCarteBleue();
```

Ordre d'appel des constructeurs :

- CompteBancaire()
- Date() ....pour l'attribut dateExpr
- CompteBancaireCarteBleue()

Ordre des destructeurs :

- CompteBancaireCarteBleue()
- Date() ....pour l'attribut dateExpr
- CompteBancaire()

Initialisation des attributs

Dans le corps des attributs : sémantique d'affectation

```
CompteBancaireCarteBleue::CompteBancaireCarteBleue(float s, int n, Date d) {
this->solde = s;
this->numCB = n;
this->dateExp = d;
}
```

Dans l'en-tête du constructeur : sémantique de copie

```
CompteBancaireCarteBleue::CompteBancaireCarteBleue(float s, int n, Date d) :
CompteBancaire(s), numCB(n), dateExp(d) { }
```

Sommaire

Liaison

- Liaison dynamique `virtual`
  préconisée pour l'objet
- Liaison statique sans indication dans le code
  pas d'extensions

illustration - débit de frais

```
//..... CompteBancaire.h .......
class CompteBancaire
{
private:
  float solde;
  static float fraisGestion;
public:
  ...
  virtual void changeAvec(float f);

  virtual float getSolde()const;

  static float getFraisGestion();

  virtual void debitFrais();
 };
```

```
//..... CompteBancaire
float CompteBancaire::fraisGes


void CompteBancaire::changeAvec
        {solde+=f;}
float CompteBancaire::getSolde
        {return solde;}
float CompteBancaire::getFrais
{return fraisGestion;}
void CompteBancaire::debitFra
{changeAvec(-fraisGestion);}
```

illustration - débit de frais

```
//..... CompteBancaireCarteBleue.h .......
class CompteBancaireCarteBleue : public virtual CompteBancaire
{
private:
  ...
  static float fraisCB;
public:
  ...

  virtual void debitFrais();
};
```

```
//..... Com
float CompteBancaireCarteBleue
void CompteBancaireCarteBleue
{changeAvec(-(getFraisGestion(
 +fraisCB));}
```

illustration - débit de frais

```
CompteBancaire **dossierComptes = new CompteBancaire*[2];
dossierComptes[0] = new CompteBancaire(200);
dossierComptes[1] = new CompteBancaireCarteBleue(74,1212,d);

for (int i=0; i<2; i++)
  dossierComptes[i]->debitFrais();
```

illustration - débit de frais

Avec `virtual`

`dossierComptes[0]->debitFrais();`
$\longrightarrow$ *appel de `debitFrais` de CompteBancaire*
`dossierComptes[1]->debitFrais();`
$\longrightarrow$ *appel de `debitFrais` de CompteBancaireCarteBleue*

Sans `virtual`

`dossierComptes[0]->debitFrais();`
$\longrightarrow$ *appel de `debitFrais` de CompteBancaire*
`dossierComptes[1]->debitFrais();`
$\longrightarrow$ *appel de `debitFrais` de CompteBancaire !!!*

Pas de liaison dynamique dans les constructeurs

```
CompteBancaire::CompteBancaire(float s){solde=s; debitFrais();}

CompteBancaireCarteBleue::CompteBancaireCarteBleue(float s, int n,
Date d) :  CompteBancaire(s){
numCB=n; dateExp=d}
```

Les deux constructeurs suivants exécutent `debitFrais` de CB

```
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(74,1212,d);
```

Explication (discutable) : Pendant l'exécution du constructeur, l'objet n'est pas encore tout à fait construit et ne serait pas en mesure d'exécuter correctement un comportement qui nécessite qu'il soit intégralement initialisé.

Désignation explicite de méthodes

Se base sur l'opérateur de portée ::

```
void CompteBancaireCarteBleue::debitFrais() {
      CompteBancaire::debitFrais();
      changeAvec(-fraisCB);
}
```

Danger : appelle une méthode située plus haut dans la hiérarchie, saute une méthode plus spécialisée.

Redéfinition de méthode

Règle de redéfinition

- même nom
- même liste de types d'arguments
- type de retour
  - identique pour les types primitifs (int, float...)
  - identique ou spécialisé pour pointeurs ou référence vers classe

Exemple :

```
virtual CompteBancaire* debitFrais()  //dans CompteBancaire
virtual CompteBancaireCarteBleue* debitFrais()  //dans CompteBancair
```

Sommaire

Classes et méthodes abstraites

Classe abstraite = sans instance propre
méthode abstraite = sans corps

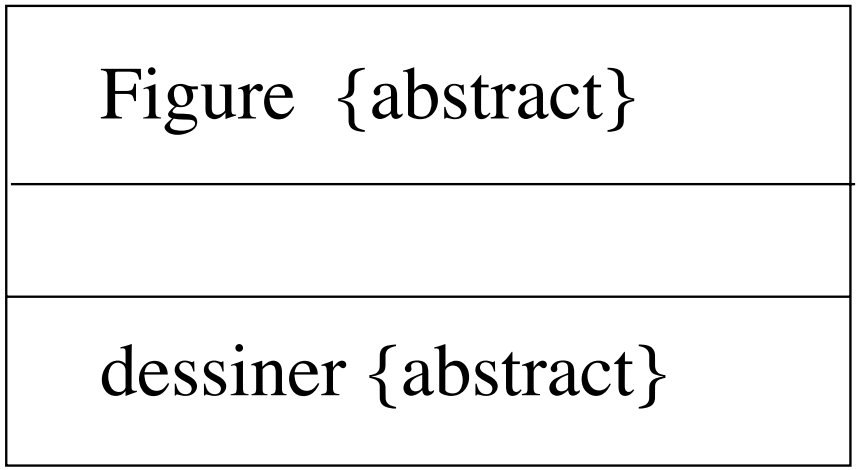


Figure: Une classe et une méthode abstraite

méthode abstraite $\Rightarrow$ classe abstraite

Classes et méthodes abstraites

En C++

Classe abstraite = pas de syntaxe particulière
méthode abstraite = méthode *virtuelle pure*

```
class Figure
{
public:
  virtual void dessine()=0;
};
```

Pas d'implémentation dans Figure.cpp
`Figure *f=new Figure();` ne compile pas !!

Sommaire

Affectations entre variables dont le type est une classe

```
CompteBancaire compteCB(10); CompteBancaireCarteBleue compteCACB(20,777,d);
compteCB = compteCACB;       //OK
compteCACB = compteCB;       // INTERDIT PAR DEFAUT .....
```

- Appelle l'opérateur `operator=`
- Sauf redéfinition, effectue une copie champ par champ (copie des champs communs)

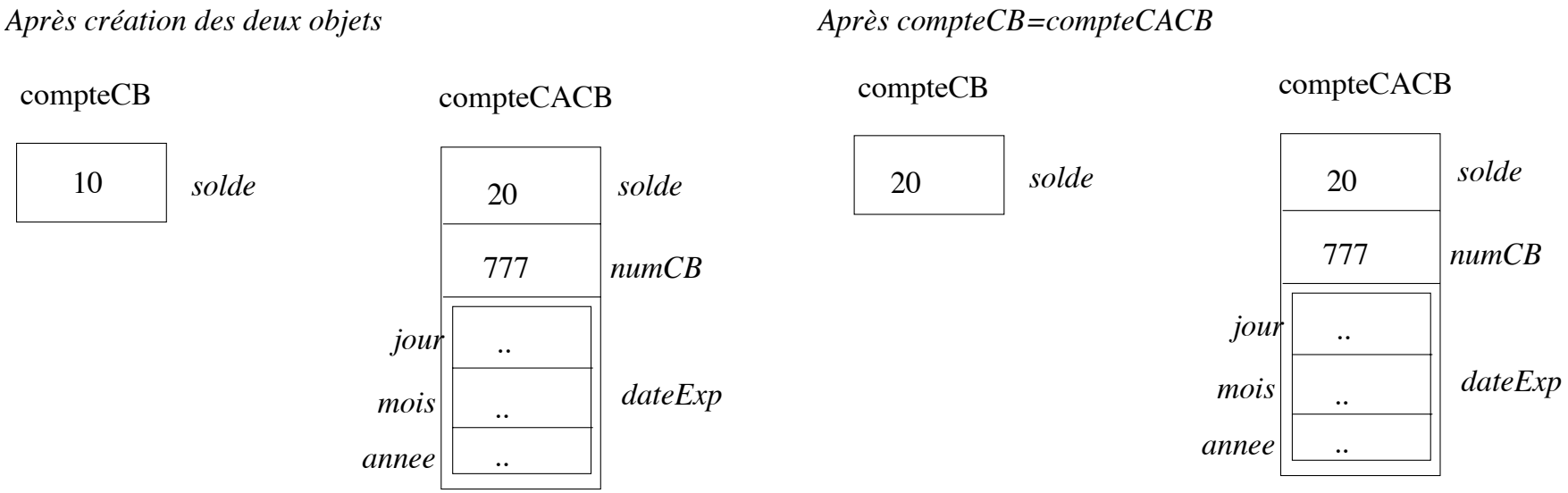


Figure: Affectation entre objets (1)

Affectations entre variables dont le type est une classe

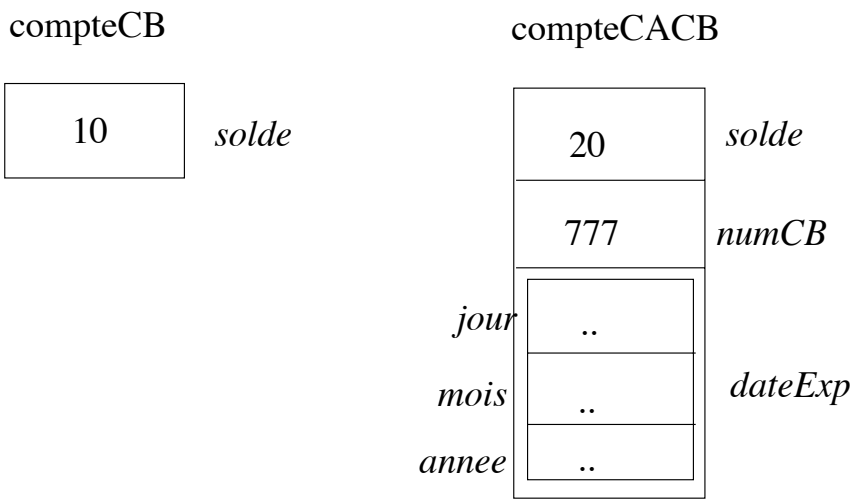
Pour rendre possible l'affectation inverse, la définir !

```
class CompteBancaireCarteBleue : virtual public CompteBancaire
{
 public:
  virtual CompteBancaireCarteBleue& operator=(const CompteBancaire&);
};
.........
CompteBancaireCarteBleue& CompteBancaireCarteBleue::operator=(const CompteB
{
  if (this != &compte)
     {changeAvec(compte.getSolde()); numCB=0; dateExp=Date();}
  return *this;
}
```

Affectations entre variables dont le type est une classe

Effet de l'affectation inverse

Après création des deux objets



Après compteCACB=compteCB

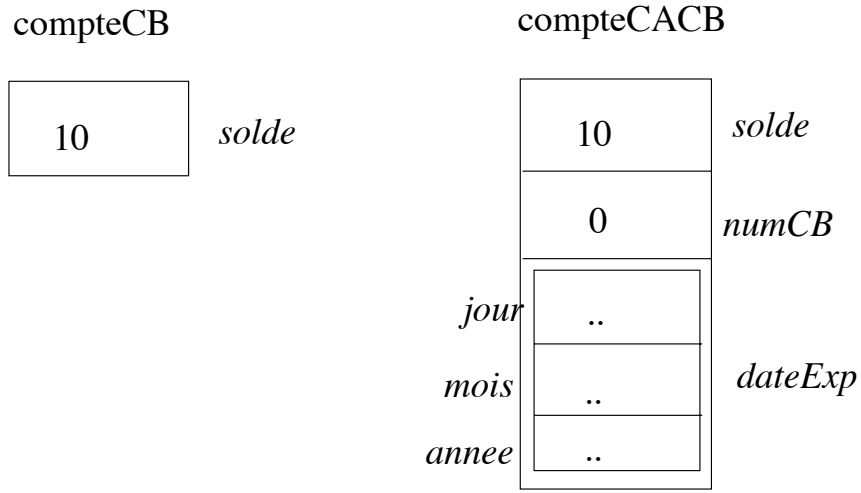


Figure: Affectation entre objets (2)

Affectation entre pointeurs (affectation polymorphe)

```
CompteBancaire *pcompteCB = new CompteBancaire(10);
CompteBancaireCarteBleue *pcompteCACB = new CompteBancaireCarteBleue(20,777,d);
pcompteCB = pcompteCACB;              //OK
pcompteCACB = pcompteCB;          //INTERDIT ....
```

Pourtant la deuxième peut avoir du sens dans certaines situations

Affectation entre pointeurs (affectation polymorphe)

L'opérateur `dynamic_cast`

- effectue une vérification de type à l'exécution
- retourne `NULL` si la coercition est incorrecte (le pointeur n'est pas du type attendu)
- existe aussi pour les références, en cas d'erreur une exception est signalée

```
CompteBancaire *pcompteCB = new CompteBancaireCarteBleue(10,333,d);
CompteBancaireCarteBleue *pcompteCACB = new CompteBancaireCarteBleue(20,7
pcompteCACB = dynamic_cast<CACB*>(pcompteCB);
```

Affectation entre pointeurs (affectation polymorphe)

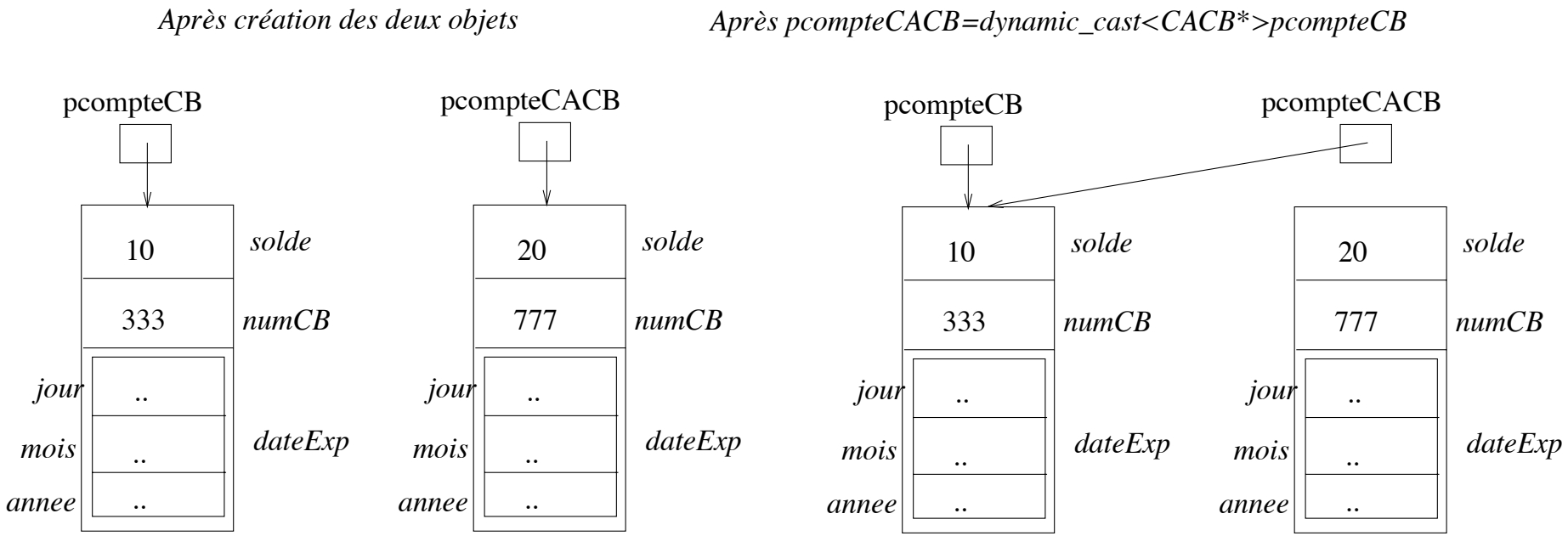


Figure: Affectation entre pointeurs vers des objets

Sommaire

Première approche de la protection

- `public`, `protected` et `private`
- se placent sur les propriétés (attributs, méthodes)
- se placent sur la déclaration d'héritage

Dans une approche simplifiée

- les propriétés publiques sont accessibles partout
- les propriétés privées d'une classe C seulement dans les méthodes de C
- les propriétés protégées de C sont accessibles pour les sous-classes de C dans les méthodes
- la déclaration sur l'héritage s'ajoute à celle de la propriété héritée