Spécialisation/généralisation et héritage simple

Programmation par objets 2, GLIN603

Marianne Huchard

29 janvier 2014

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Classification et approches à objets

Rapprocher monde réel et représentation informatique
Idée 1-

- se baser sur la notion de « concept »
  - *extension* l'ensemble des objets couverts par le concept
  - *intension* l'ensemble des prédicats vérifiés par les objets couverts
  - classes et interfaces en Java ; classes en C++

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
Idée 2-

- classification / organisation des concepts par spécialisation
  - inclusion des *extensions*
  - raffinement des *intensions*
  - représenté dans les langages par l'héritage

Exemple

Le concept de « carré » spécialise le concept de « rectangle »
l'ensemble des carrés est inclus dans l'ensemble des rectangles (inclusion des extensions).
les propriétés du concept « rectangle » s'appliquent au concept « carré » et se spécialisent (raffinement des intensions), ex. largeur et hauteur sont égales par exemple, le calcul du périmètre et de l'aire sont simplifiés, etc.

Classification et approches à objets

Formes de la classification

- arborescence, héritage simple (Smalltalk, Java-classes)
- graphe sans circuit, héritage multiple (Eiffel, C++, Java-interfaces)

Dans ce cours : héritage simple en C++

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Cas d'étude

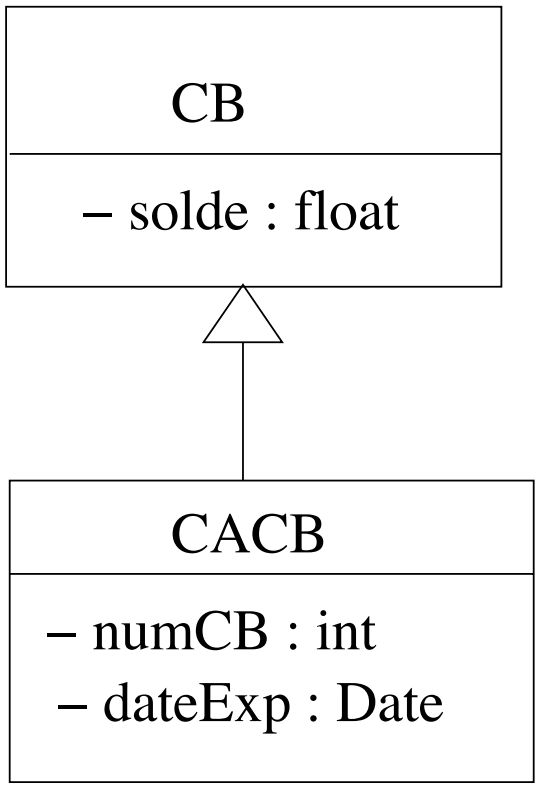

Figure : Comptes bancaires simplifiés

Déclaration dans les fichiers *header*

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

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

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Ordre d'appel

- les constructeurs des classes sont appelés du haut vers le bas de la hiérarchie d'héritage (depuis les super-classes vers les sous-classes)
- les constructeurs sont appelés pour les attributs dont le type est une classe (pas un pointeur sur une classe), et ceci juste avant le constructeur de la classe qui déclare l'attribut
- la destruction se fait toujours en sens inverse de la construction

Ordre d'appel

`CACB *pc = new CACB();` ou `CACB c;`

Ordre d'appel des constructeurs :

CB()

Date() ... pour l'attribut dateExp

CACB()

Ordre inverse pour les destructeurs :

~CACB()

~Date() ... pour l'attribut dateExp

~CB()

Passage des paramètres

```
//...... CB.h .......
class CB {
private:
  float solde;
public:
  CB(float s);
};
```

```
//...... CB.cc

CB::CB(float s){solde=s;}
```

```
//...... CACB.h .......
class CACB : public virtual  CB{
private:
  int numCB;
  Date dateExp;
public:
 CACB(float s, int n, Date d);
};
```

```
//...... CACB.cc

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

Initialisation des attributs

Possible dans le corps du constructeur, sémantique d'affectation

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

Possible dans l'entête du constructeur, sémantique de copie

```
CACB::CACB(float s, int n, Date d)
: CB(s),numCB(n),dateExp(d)
{} .
```

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Liaison

- Liaison dynamique `virtual`
  préconisée pour l'objet
- Liaison statique sans indication dans le code
  préconisée pour les applications très sensibles au coût en temps et espace et sous la condition qu'il n'y aura pas d'extensions basées sur les mécanismes objets

illustration - débit de frais

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

  virtual float getSolde()const;

  static float getFraisGestion();

  virtual void debitFrais();
 };
```

```
//...... CB.cc

float CB::fraisGestion=10;

void CB::changeAvec(float f)
                {solde+=f;}
float CB::getSolde()const
               {return solde;}
float CB::getFraisGestion()
{return fraisGestion;}
void CB::debitFrais()
{changeAvec(-fraisGestion);}
```

illustration - débit de frais

```
//...... CACB.h .......
class CACB : public virtual  CB
{
private:
  ...
  static float fraisCB;
public:
  ...

  virtual void debitFrais();
};
```

```
//...... CACB.cc

float CACB::fraisCB=5;
void CACB::debitFrais()
{changeAvec(-(getFraisGestion()
 +fraisCB));}
```

illustration - débit de frais

```
CB **dossierComptes = new CB*[2];
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(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 CB*

dossierComptes[1]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CACB*

Sans `virtual`

dossierComptes[0]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CB*

dossierComptes[1]->debitFrais();

$\longrightarrow$ *appel de debitFrais de CB!!!*

Pas de liaison dynamique dans les constructeurs

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

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.

Appel de la méthode héritée

Mécanisme de désignation explicite qui se base sur l'opérateur de portée `::`

```
void CACB::debitFrais()
{
  CB::debitFrais();
  changeAvec(-fraisCB);
}
```

- Danger : permet d'appeler une méthode située plus haut dans la hiérarchie, en sautant 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
  - spécialisé pour pointeur ou référence vers une classe

Exemple :

`virtual CB* debitFrais()` dans `CB`

`virtual CACB* debitFrais()` dans `CACB`

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Classes et méthodes abstraites

Classe abstraite = sans instance propre
méthode abstraite = sans corps

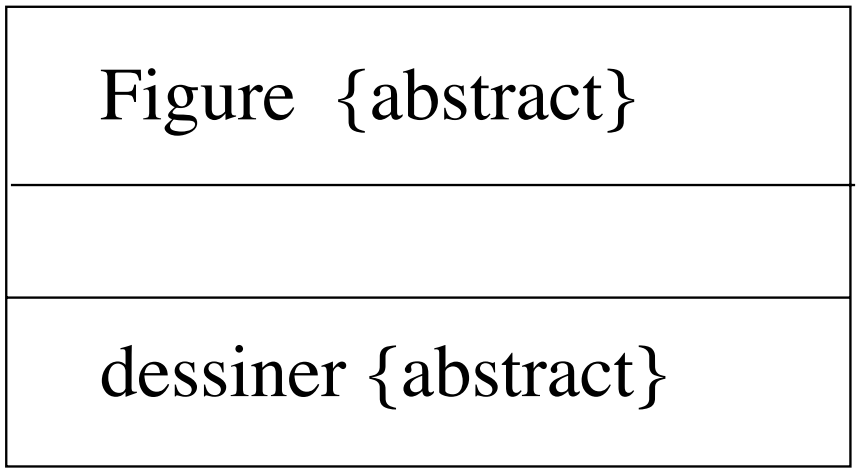

Figure : Une classe et une méthode abstraite

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.cc
`Figure *f=new Figure();` ne compile pas!!

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

Affectations entre variables dont le type est une classe

```
CB compteCB(10); CACB compteCACB(20,777,d);
compteCB = compteCACB;
// INTERDIT PAR DEFAUT ..... compteCACB = compteCB;
```

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

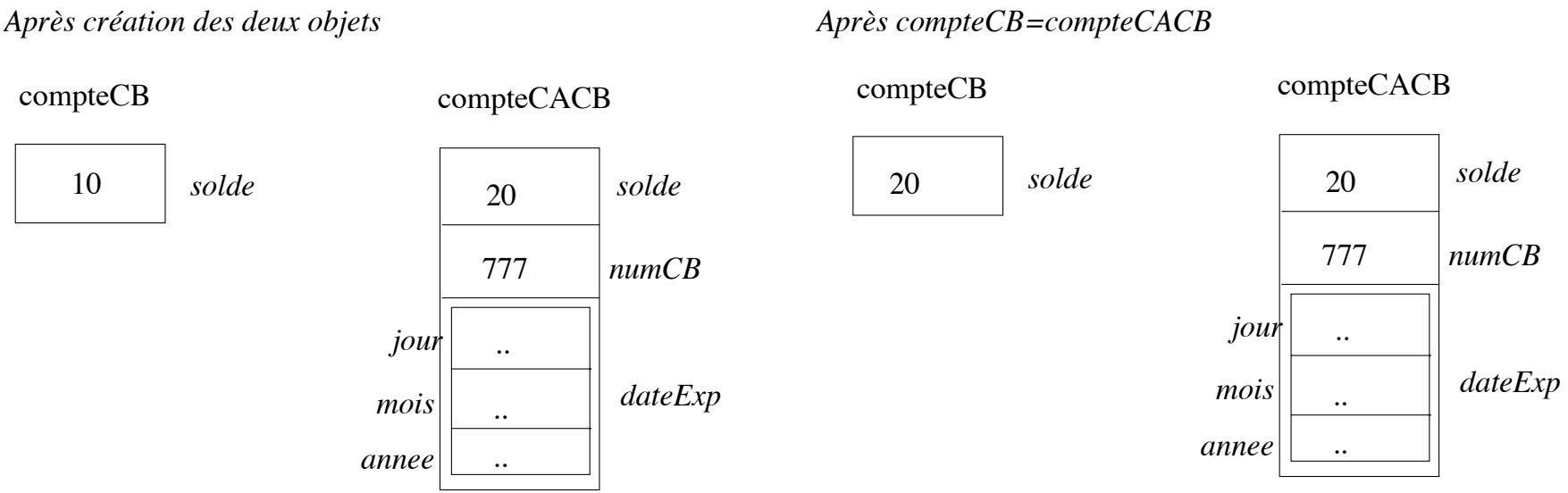

Figure : Affectation entre objets (1)

Affectations entre variables dont le type est une classe

Pour rendre possible l'affectation inverse, la définir !

```
class CACB : virtual public CB
{
 public:
  virtual CACB& operator=(const CB&);
};
........
CACB& CACB::operator=(const CB& compte)
{
  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

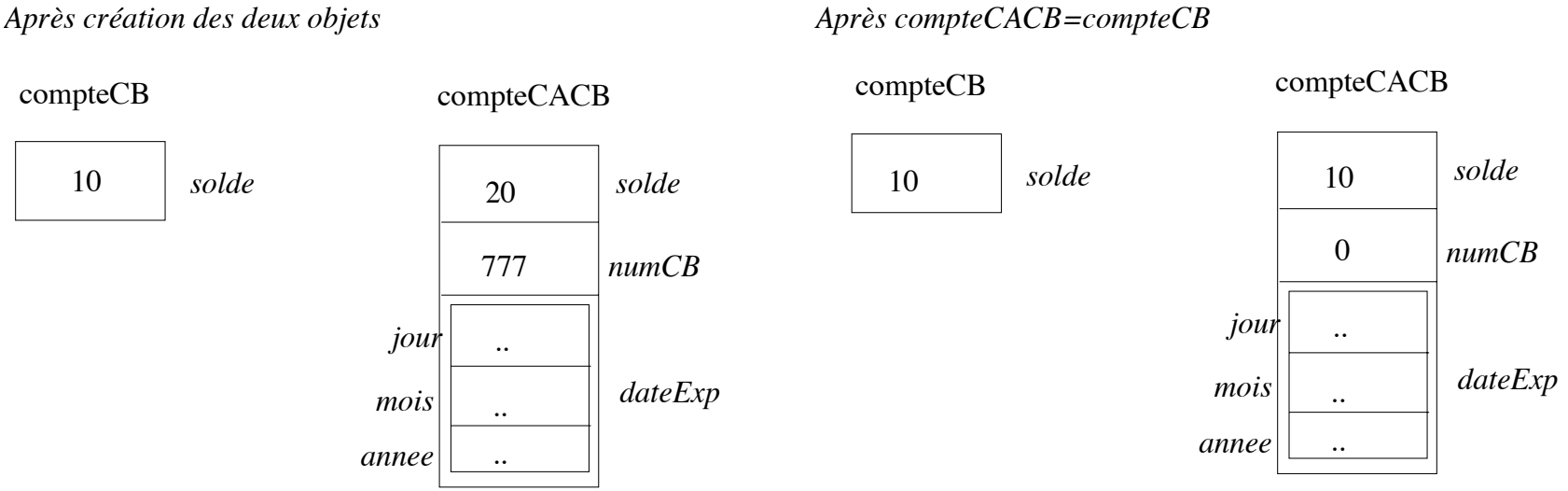

Figure : Affectation entre objets (2)

Affectation entre pointeurs (affectation polymorphe)

```
CB *pcompteCB = new CB(10); CACB *pcompteCACB = new CACB(20,777,d);
pcompteCB = pcompteCACB;
//INTERDIT ....  pcompteCACB = pcompteCB;
```

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

```
CB *pcompteCB = new CACB(10,333,d);
CACB *pcompteCACB = new CACB(20,777,d);
pcompteCACB = dynamic_cast<CACB*>(pcompteCB);
```

Affectation entre pointeurs (affectation polymorphe)

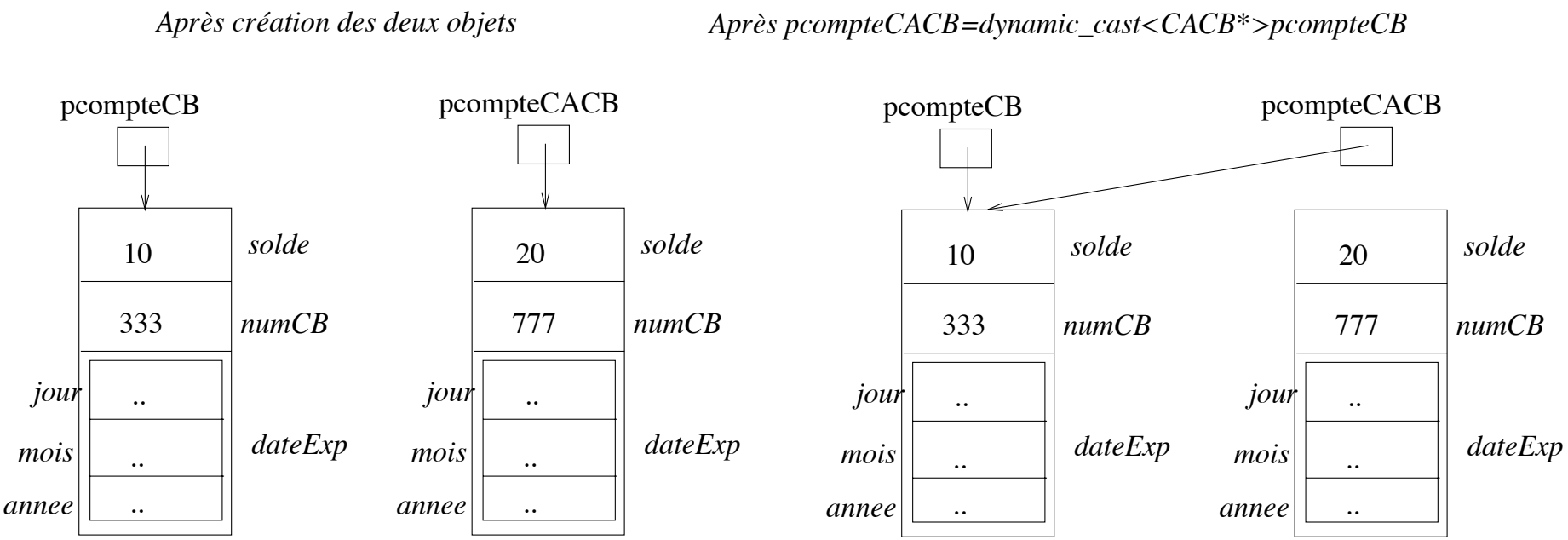

Figure : Affectation entre pointeurs vers des objets

Sommaire

Spécialisation/héritage

Déclaration

Constructeurs/destructeurs

Méthodes

Abstractions

Coercition

Protection

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 une sous-classe C' de C dans ses méthodes soit sur des instances de C' soit sur des instances des sous-classes de C' (jamais en remontant)
- la déclaration sur l'héritage s'ajoute à celle de la propriété héritée

Les détails dans un prochain cours !

Introspection et réflexivité

Introspection. capacité d'un langage à inspecter, en cours d'exécution, les éléments d'un programme, par exemple pour connaître la classe exacte de l'objet stocké dans une variable ou encore la liste des attributs d'une classe

Réflexivité. les éléments du programme peuvent être véritablement manipulés ; par exemple on peut créer une nouvelle classe pendant l'exécution d'un programme ou encore instancier une classe en donnant seulement son nom sous forme d'une chaîne de caractères.

En C++ : RTTI

RTTI. Run Time Type Information

- opérateur `typeid`, retourne pour un objet *o*, une instance de `type_info` qui décrit le type (la classe) de *o*.
- `type_info`
  - opérateurs $!=$ et $==$ pour comparer les types de deux objets
  - méthode `name` qui retourne une chaîne contenant le nom de la classe de l'objet.

Affichage du nom de la classe `CACB`

```
CB *pcb=new CACB();
cout << typeid(*pcb).name() << endl;
```