Héritage et Polymorphisme

- classes et héritage
- héritage et droit d’ accès
- héritage et constructeurs
- héritage et redéfinition
- polymorphisme
- méthodes virtuelles

Apr 25, 2001 13:01

Programmation Orientée Objets

Répétition: « les 4 aspects essentiels »

- ***encapsulation*** *objets*
  (données ↔ comportement)
  (public ↔ private)
- ***abstraction*** *classes*
  (↔ instanciation)
- ***héritage*** (↔ extensibilite, réutilisabilite)
- ***polymorphisme*** (↔ réutilisabilite)

Héritage (I)

Technique pour ***enrichir*** *des* ***classes***:

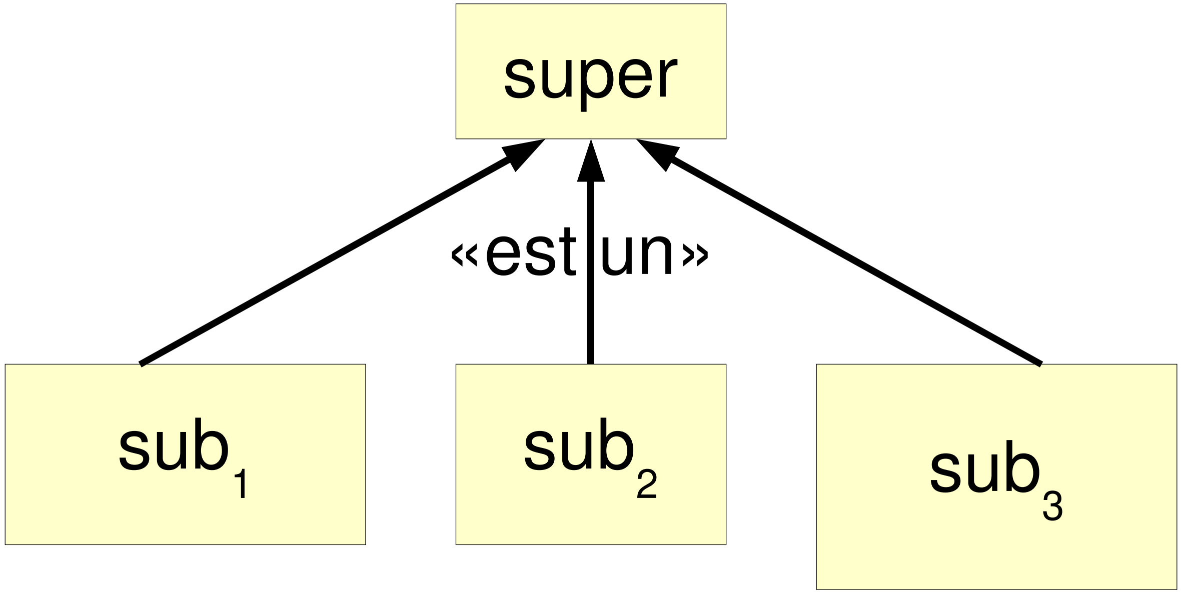
sur–classes (parentes, classe de base):

plus générales, **moins riches**

sous–classes (enfants, classe dérivée):

plus spécifiques, **plus riches**

création
des **sub**
à partir
des **super**



Héritage (II)

Une ***sous−classe***

- ***hérite tous les membres de la sur−classe*** (l'ensemble des attributs et des méthodes),

mais elle peut aussi

- ***redéfinir*** les attributs herite, et
- ***rajouter*** des membres supplémentaires.

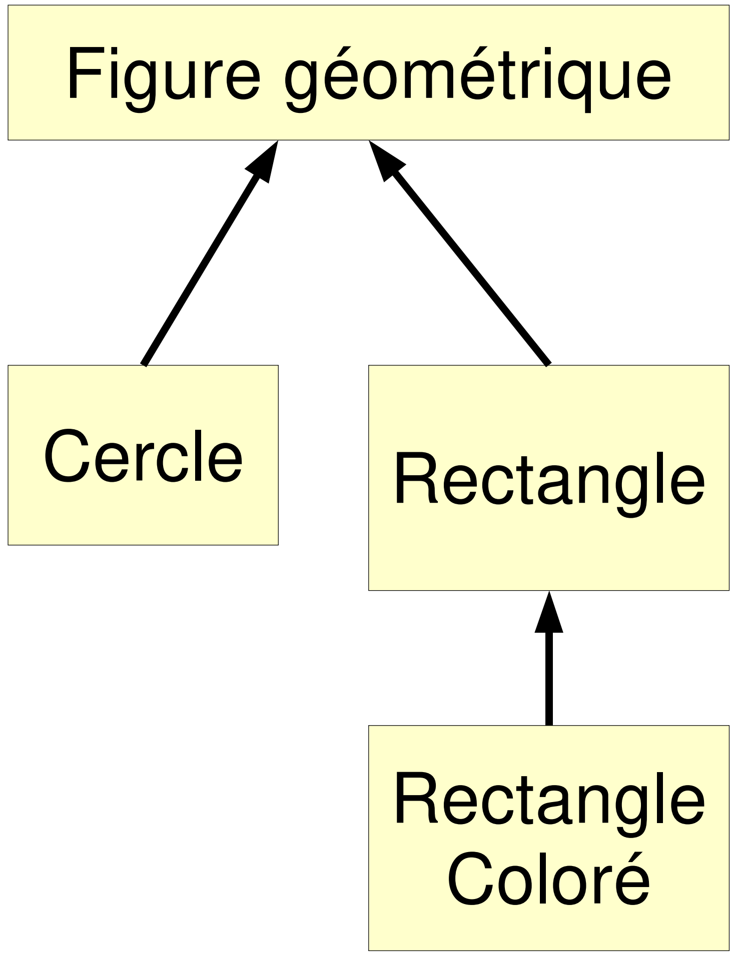
Consequences:

- ✗ relations structurelles explicites entre classes
- ✗ réduction de rédondance (description, stockage, ...)

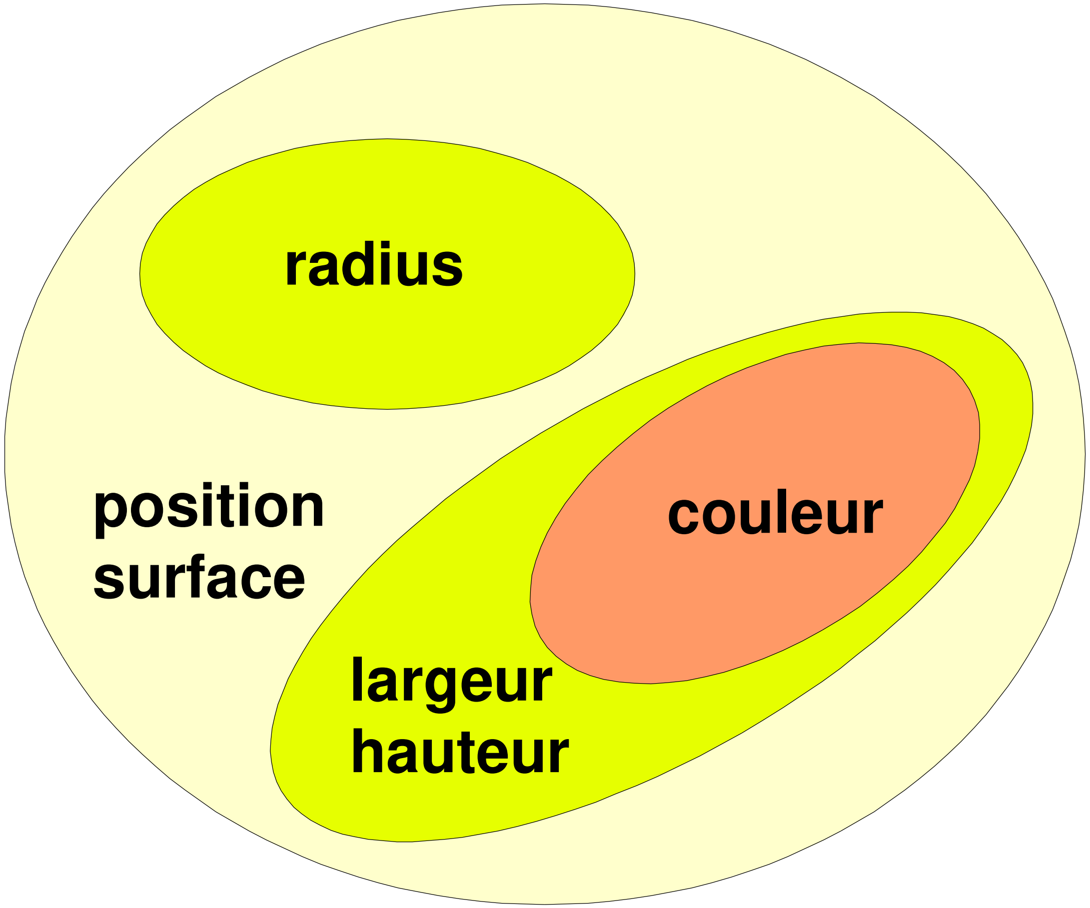
Exemple

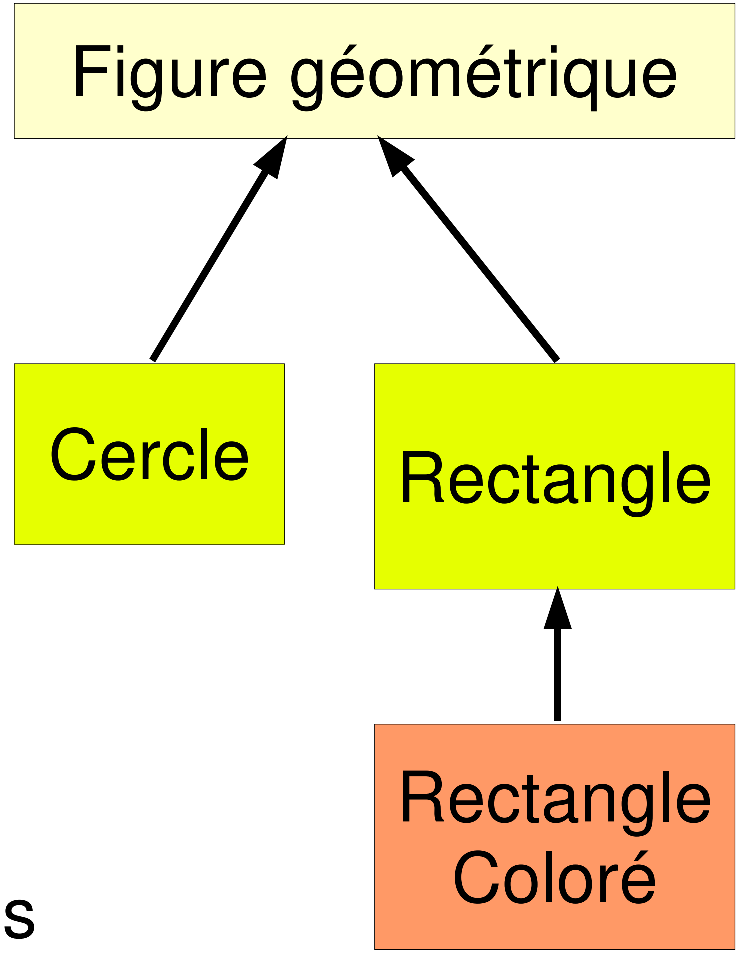
Représentation des rectangles comme sous–classe des figures ...

Enrichissement des rectangles (avec largeur, hauteur, et surface) par un attribut couleur.



Sous–Ensembles et Sous–Classes





chaque élément possède les membres
de tous ses sur–ensembles

Hiérarchie de Classes

L'héritage mène à un réseau de dépendances entre les classes.

- graphe d'héritage
- arborescence

Problèmes ? Pièges ?

- droit d'accès
- constructeurs
- redéfinition avec typage ...
- héritage simple et héritage multiple

Syntaxe

```
class NomSousClasse : nomSurClasse
{
  // placer ici uniquement la déclaration
  // 1) des membres supplémentaire
  //    (données ou fonctions)
  // 2) des membres redéfinis
  //    (données ou fonctions)
  // 3) des nouveaux con-/de-structeurs

};
```

Exemple: Rectangle Coloré

```
enum color {red,green,blue};

class Rect {
public:
  int haut, larg;
};

class ColRect : Rect {
public:
  color col;
};

Rect r;
ColRect c;
```

Exemple: Employé (I)

```
class Employe {
public:
  Employe(const char *nm);
  char *getNom() const;
private:
  char nom[30];
};
```

caractéristiques communes à tous les employés

Exemple: Employé (II)

```
class Temporaire : public Employe {
public:
   Temporaire(const char *nm);
   void setTarif (float trf);
   void setHeures(float hrs);
private:
   float tarif;
   float heures;
};
```

caractéristiques spécifiques pour les employés temporaires (payé à l'heure)

Exemple: Employé (III)

```
class Cadre : public Employe {
public:
   Cadre(const char *nm);
   void setSalaire (float sal);
private:
   float salaire;
};
```

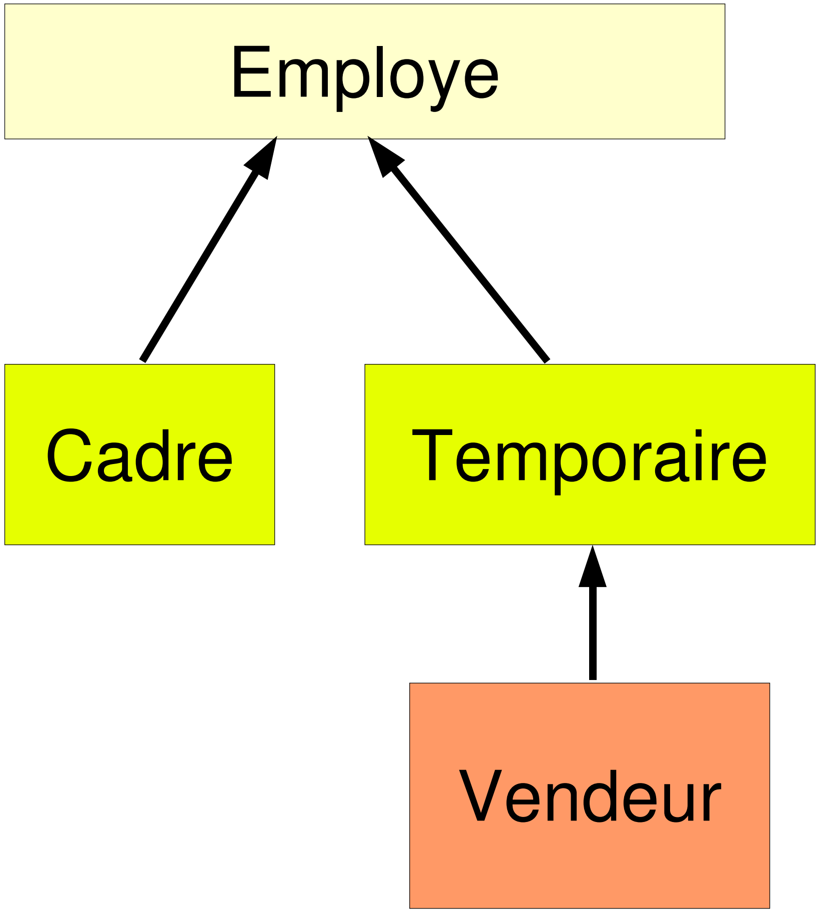
caractéristiques spécifiques pour les cadres (salaire fixe)

Exemple: Employé (IV)

```
class Vendeur : public Temporaire {
public:
   Vendeur(const char *nm);
   void setCommission (float sal);
   void setVentes (float ven);
private:
   float salaire;
   float ventes;
};
```

caractéristiques spécifiques pour les cadres (salaire fixe)

Exemple: Employé (V)



- accès aux membres ?
- constructeurs ?
- redéfinition ?

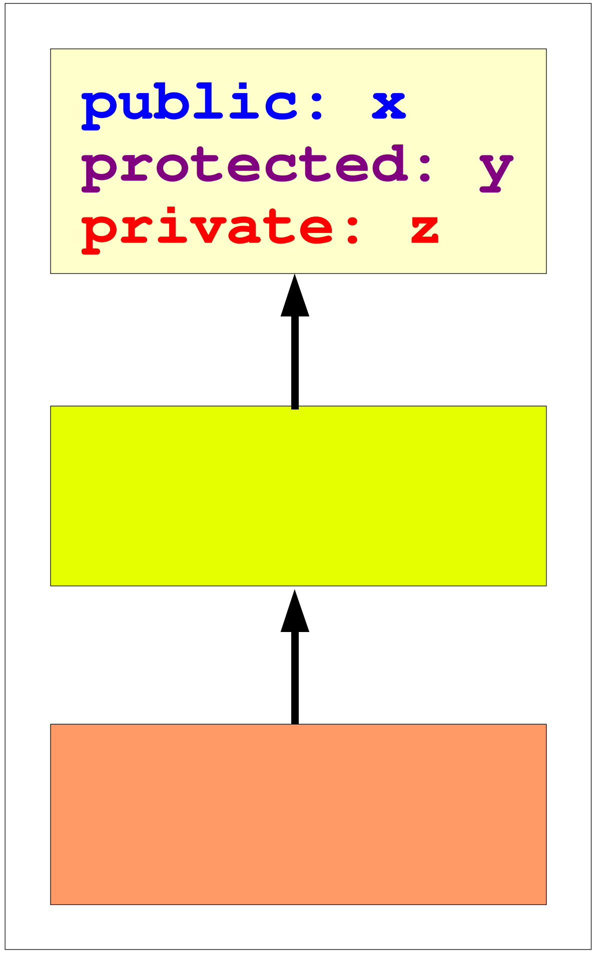
une sous–classe
ne peut utiliser que
l'interface accéssible
de sa sur–classe

Droit d'Accès
Masquage d'Information

`public` (*interface de communication*)
visibilité totale

`protected` (*interface de programmation*)
visibilité restreinte
aux méthodes de la classe
et de sa descendance

`private`
visibilité strictement restreinte
aux méthodes de la classe



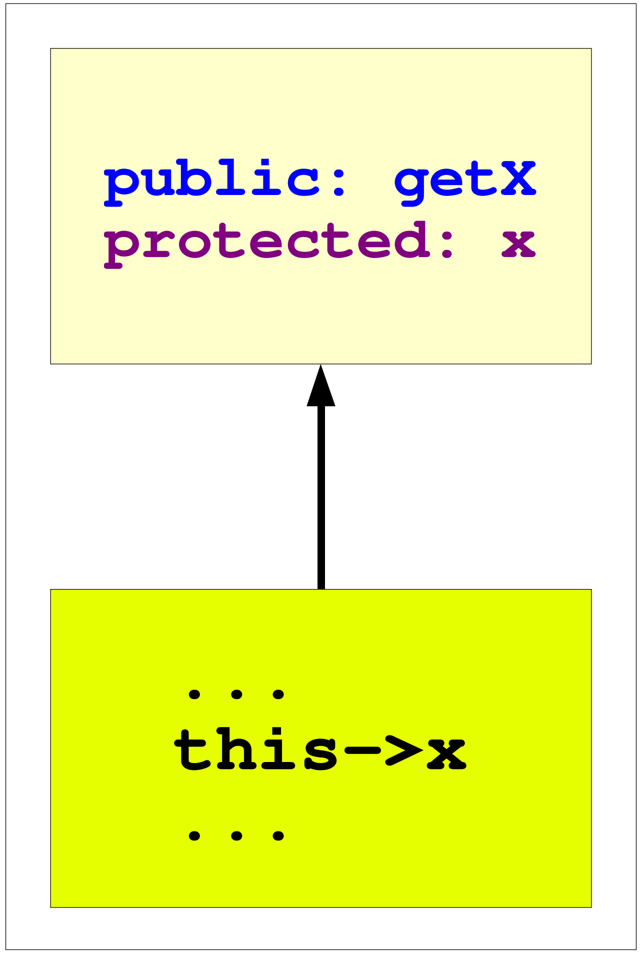
Syntaxe

```
class NomClasse [: nomSurClasse ]
{
  // par défault: niveau private

  public:
  // membres au niveau public
  protected:
  // membres au niveau protected
  private:
  // membres au niveau private

};
```

aucun ordre est imposé, mais quand même conseillé ...

Exemple

```
class B {
public:
  int getX()
  { return x; };
protected:
  int x;
};

class A : public B {
public:
  void setX(const int x)
  { this->x = x; };
};
```



Héritage et Droit d'Accès

La sous–classe

- peut ***conserver*** ou ***restreindre***, mais
- ne peut ***pas relâcher***

les droits d'accès.

Syntaxe:

```
class NomClasse [: [droit] nomSurClasse ]
{...}
```

accés initial \ *héritage*	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	–	–	–

Héritage et Constructeurs

Principe: L'initialisation se fait à l'endroit où l'attribut est explicitement défini. Par conséquent, la recherche des constructeurs se fait le long l'hiérarchie des classes.

```
class SousClasse : [droit] SurClasse
{ ...
  SousClasse (<args>)
    : SurClasse(<args>),
      <attribut>(<valeur>), ...
  {
    // corps du constructeur
  }
  ...
};
```

ne pas (toujours) obligatoire

Exemple (I)

```
enum color {red,green,blue};

class Rect {
public:
   int haut, larg;
   Rect (int i): haut(i), larg(i) { };

class ColRect : public Rect {
public:
   color col;
   ColRect (): Rect(2), col(red) { }
};
```

NB: ici obligatoire !

```
Rect r(3);
ColRect c;
```

Exemple (II)

```
enum color {red,green,blue};

class Rect {
public:
  int haut, larg;
  Rect (): haut(4), larg(2) { };
};

class ColRect : public Rect {
public:
  color col;
  ColRect (): col(red) { }
};

Rect r;
ColRect c;
```

NB: pas d'appel explicite au constructeur de la surclasses

Héritage et Redéfinition

Si l’on redéfinit une méthode **m** dans une sous–classe **A**,
qui est alors déjà connue dans la classe de base **B**,
que se passe–t–il quand on appelle la methode **m**
à une instance de la sous–classe **A**?

C’est le code de la méthode redéfinie qui est utilisé.

Ne peut–on plus accéder a la méthode de la sur–classe?

Oui, on le peut faire en utilisant l’opérateur **::** ...

Exemple

```
class B {
protected:
   void f(int i);
};

class A : public B {
protected:
   void f(int x);
};

void B::f(int i) { return i*i }
void A::f(int x) { return B::f(x*2) }

A a;
a.f(1);
```

Polymorphisme

« le caractère qui revêt plusieurs formes »

polymorphisme des fonctions (ad−hoc):

surcharge permet d'utiliser le même nom
pour différentes fonctions

polymorphisme des données (universel):

d'inclusion/sous−typage
suivant la hiérarchie des classes

paramétrique
indépendant au type des arguments

Substitutivité

« Une instance d’une sous–classe
peut etre utilisée dans chaque context
où une instance de la sur–classe est attendue. »

Exemple:
Soit **A** une sous–classe de **B**.
Soit **a** une instance de **A**.
Soit **f** une fonction qui prend des arguments de type **B**.
Alors, **f**(**a**) est aussi bien définie, parce que
tout ce que la fonction **f**
pourrait faire avec (ou extraire d’) une instance de classe **B**,
elle peut aussi faire avec une instance de classe **A**.

Substitutivité

En fait, par transitivité, chaque instance d'une sous–classe est aussi une instance de la sur–classe, et aussi une instance de la sur–classe de celle–ci, et cetera ...

Exemple: Substitutivité (I)

```
class A {
public:
  int a;
  ...
{;
class B : public A {
public:
  int b;
  ...
};
```

```
void affice(A obj) {
  cout << "Etat: ";
  cout << obj.a << endl;
};
int main() {
  B bb; bb.a=4; bb.b=3;
  A aa; aa=bb;
  affiche(aa);
  affiche(bb);
};
```

Exemple: Substitutivité (II)

```
class A {
public:
  int a;
  void dump(){
    cout<<", a="<<a; }
  ...
{;
class B : public A {
public:
  int b;
  void dump(){
    A::dump();
    cout<<", b="<<b; }
  ...
};
```

```
void affice(A obj) {
  cout << "Etat: ";
  cout << obj.a << endl;
};
int main() {
  B bb; bb.a=4; bb.b=3;
  A aa; aa=bb;
  affiche(aa);
  affiche(bb);
  bb.dump;
};
```

Exemple: Substitutivité (III)

```
class A {
public:
  int a;
  void dump(){
    cout<<", a="<<a; }
  ...
{;
class B : public A {
public:
  int b;
  void dump(){
    A::dump();
    cout<<", b="<<b; }
  ...
};
```

```
void affice(A &obj) {
  cout << "Etat: ";
  cout << obj.a << endl;
};
int main() {
  B bb; bb.a=4; bb.b=3;
  A aa; aa=bb;
  affiche(aa);
  affiche(bb);
  bb.dump;
};
```

Liaisons statiques et dynamiques

statique (static binding)

la signature de la fonction
connu au moment de la compilation
détermine la version de la fonction à exécuter
$\Rightarrow$ *surcharge*

dynamique (late binding)

la version de la fonction à exécuter
n'est pas connu jusqu'au moment de l'exécution

Méthodes virtuelles

le mot–clef **`virtual`** indique au compilateur qu'une méthode peut faire l'objet d'une resolution dynamique des liens.

```
virtual <type_retour> <nom>(<args>);
```

Les redéfinitions sont aussi considéré comme virtuelles.

Constructeurs ne peuvent pas etre virtuels.

Exemple: Méthodes Virtuelles

```
class A {
public:
   int a;
   virtual void dump(){
      cout<<", a="<<a; }
   ...
{;
class B : public A {
public:
   int b;
   void dump(){
      A::dump();
      cout<<", b="<<b; }
   ...
};
```

```
void affice(A &obj) {
   cout << "Etat: ";
   cout << obj.a << endl;
};
int main() {
   B bb; bb.a=4; bb.b=3;
   A aa; aa=bb;
   affiche(aa);
   affiche(bb);
   bb.dump;
};
```