Héritage

1. Objectif
2. Classe composée ou classe dérivée
3. Syntaxe
4. Accès aux données et méthodes
5. Classes dérivées et constructeurs
6. Héritage multiple
7. Polymorphisme
8. Méthodes virtuelles
9. Un exemple: les expressions
10. Evaluation de polonaise postfixée, sans arbre
11. Table de fonctions virtuelles

Objectif

L'héritage est l'un des principes fondamentaux de la programmation objet. Il a pour objectif de hiérarchiser les classes et les objets.

L'héritage est mis en oeuvre par la construction de classes dérivées. Une classe dérivée

* contient les données membres de sa classe de base;
* peut en ajouter de nouvelles;
* possède a priori les méthodes de sa classe de base;
* peut redéfinir (masquer) certaines méthodes;
* peut ajouter de nouvelles méthodes.

L'héritage peut être simple ou multiple.

Avantages:

* Une classe dérivée modélise un cas particulier de la classe de base, et peut donc utiliser la classe de base;
* l'héritage encourage la recherche de la généralité et de la généricité;
* une hiérarchie de classes facilite la solution de problèmes complexes;
* facilite la maintenance, le développement et les extensions;
* peut ajouter de nouvelles méthodes.

Composition ou Héritage

Une classe est composée si certains de ses membres sont eux-mêmes des objets.

```
class DescrHabitant {
protected:
  string nom;
  string domicile;
};
```

```
class Carte {
  DescrHabitant h;
  int age;
  float age;
};
```

La description d'un habitant fait partie d'une carte d'identité; correct car une carte d'identité n'est pas un cas particulier d'un habitant.

```
class DescrResident : public DescrHabitant{
protected:
  string residence;
};
```

La description d'un résident est plus complète que celle d'un habitant: l'information supplémentaire est l'adresse de sa résidence (qui peut être différente de celle de son domicile).

[Mon Conseil : privilégier la composition ou la délégation]

Syntaxe

La syntaxe pour la définition d'une classe dérivée est

```
class classe_derivee : protection classe_de_base
  {...};
```

Les types de protection sont public, protected, private.

En général, on choisit public.

Remarquer que l'on a un lien très fort entre la classe dérivée et la classe de nase qui est très difficile de casser, c'est pourquoi on privilégie la composition a l'héritage.

Des Exemples

* Une voiture comporte quatre (ou cinq) roues, donc

```
class Automobile {
  Roue roues[5];
  ...
}
```

* Un segment comporte deux points;
* Un anneau est un cas particulier d'un cercle;
* Une forme contient un rectangle englobant:

```
class Shape {
  Rect r;
  ...
}
```

* Un triangle est une forme particulière:

```
class Triangle : public Shape {
  ...
}
```

Discussion : un carré est-il un cas particulier d'un rectangle ?

Accès aux données et méthodes

Les attributs de la classe de base peuvent être désignés directement ou par le nom de la classe de base, en employant l'opérateur de portée ::

```
class DescrHabitant {
  string nom;
  string domicile;
public:
  DescrHabitant();
  void show();
};

class DescrResidence : public DescrHabitant {
  string residence;
public:
  DescrResidence();
  void show();
};
```

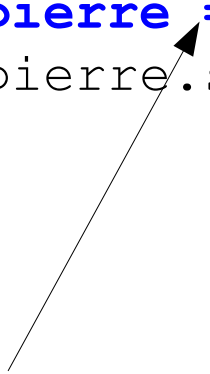
Implantations

```
DescrHabitant::DescrHabitant() {
  cout << "Entrez votre nom : "; cin >> nom;
  cout << "Entrez votre domicile : "; cin >> domicile;
}
void DescrHabitant::show() {
  cout << "Nom: \t\t" << nom << endl;
  cout << "Domicile \t" << domicile << endl;
}

DescrResidence::DescrResidence() {
//  appel implicite du constructeur par défaut de la classe de base.
  cout << "Entrez votre residence : "; cin >> residence;
}
void DescrResidence::show() {
  DescrHabitant::show(); // l'appel de la méthode de la classe de base.
  cout << "Residence \t" << residence << endl;
}
```

Le principe de substitution

```
void main()
{
   DescrHabitant pierre;
   pierre.show();
   DescrResidence paul;
   paul.show();
   pierre = paul;
   pierre.show();
}
```



```
Entrez votre nom : Dupont
Entrez votre domicile : Paris
Nom:          Dupont
Domicile       Paris
Entrez votre nom : Duval
Entrez votre domicile : Marne
Entrez votre residence : Champs
Nom:          Duval
Domicile       Marne
Residence      Champs
Nom:          Duval
Domicile       Marne
```

Le principe de substitution :

Le principe de substitution s'énonce comme suit: toute occurrence d'un objet de la classe de base peut toujours être remplacée par un objet d'une classe dérivée.

Ici l'opérateur d'affectation prend un DescrHabitant en second paramètre.

le masquage

La redéfinition d'une méthode dans une classe dérivée est le masquage. Dans

```
class Base { ...
  public : void f(char* c);...
};
classe Der : public Base {
  public : void f(int i);...
};
```

il y a masquage, et non surcharge, même si les signatures des méthodes sont différentes.
Les fonctions ne sont pas dans le même espace de "portée" (donc dans des espaces de nom différents).

Pour un objet `d` de la classe `Der`,

```
int i;...  d.f(i);
```

désigne la méthode de `Der::f(int)`.

L'appel

```
d.f("Ben"); // erreur
```

est rejeté, car il n'y a pas de méthode `Der::f(char*)`. On peut par contre écrire

```
d.Base::f("Ben");
```

qui permet l'appel.

La construction d'un objet d'une classe dérivée comporte

* la réservation de la place pour l'objet; // philosophiquement oui !
* l'appel d'un constructeur approprié (constructeur par défaut) de la classe de base;
* appel des constructeurs pour les membres objets;
* l'exécution du corps de la fonction constructeur.

Un constructeur autre que celui par défaut est indiqué dans la liste d'initialisation.

```
class Animal {
  protected:
    string nom;
  public:
    Animal(string n) : nom(n) {}
    void show()
     {cout << nom <<endl;}
};
```

```
class Ours: public Animal {
    string couleur;
  public:
    Ours(string n, string c) :
        Animal(n), couleur(c) {}
    void show()
       { cout << nom <<' '<< couleur << endl; }
};
```

```
void main()
{
  Animal a("Elephant"); a.show();
}
```

```
Ours o("Panda","blanc"); o.show();
```

La sortie est
Elephant
Panda blanc

* La méthode show() de la classe dérivée masque la méthode de la classe de base, mais
* Dans une fonction, un paramètre formel objet de la classe de base peut être substitué par un objet de la classe dérivée (principe de substitution). Il y a conversion de l'objet dérivé en objet de la classe de base.

Ainsi

```
o.Animal::show();
```

affiche simplement `Panda`. Définissons une fonction globale

```
void affiche(Animal* a) {
  cout <<"Nom : "; a ->show();
}
```

Le résultat est

```
Nom : Panda
```

car il y a `conversion` de type. (Si, en revanche, la méthode `show()` de la classe `Animal` est définie `virtual`, la conversion n'a pas le même effet.)

Héritage multiple

Une classe peut hériter de plusieurs classes. Les classes de bases sont alors énumérées dans la définition.

```
class D : public B1, public B2 {...}
```

L'héritage multiple est utile dans l'emploi d'interfaces (voir plus loin).

Héritage Multiple - liste chaînées d'entiers

Une liste chaînée comporte deux parties de nature différente, les données et le chaînage.
Une liste ``pure'' est formée de cellules dont chacune ne contient qu'un pointeur vers la suivante. Le constructeur insère une cellule.

```
typedef class Cellule* Liste;
class Cellule {
  protected:
    Liste suiv;
  public:
    Cellule(Liste a) : suiv(a) {}
};
```

Un ``Entier'' a une donnée membre unique.

```
class Entier {
  protected:
    int val;
  public:
    Entier(int i) : val(i) {}
    void show() { cout << val << ' ';}
};
```

Une classe de listes d'entiers dérive des deux classes:

```
typedef class CelluleEntiers* ListeEntiers;
class CelluleEntiers : public Cellule, public Entier {
public:
  CelluleEntiers(Liste a, int i) : Cellule(a),Entier(i) {}
  void show();
};
```

La classe a deux champs

* le champ int val hérité de la classe Entier;
* le champ de Liste suiv hérité de la classe Liste.

La fonction d'affichage dépend des deux classes de bases:
* de Entier pour l'affichage des éléments;
* de Cellule pour le chaînage.

Elle s'écrit

```
void CelluleEntiers::show() {
  Entier::show();
  if (suiv)
    ((ListeEntiers) suiv) -> show();
};
```

La conversion de `suiv` du type `Liste` en type `ListeEntiers` est nécessaire pour la cohérence.

```
void main() {
  ListeEntiers a = 0; int j;
  for (int i = 0; i < 5; i++) {
    cin >> j;
    a = new ListeEntiers(a, j);
  }
  a -> show();
}
```

Polymorphisme

L' édition de lien est l'opération qui associe, à l'appel d'une fonction, l'adresse du code de la fonction.

On distingue

* La liaison statique (ou précoce): effectuée à la compilation
* La liaison dynamique (ou tardive) : effectuée à l'exécution.

Le polymorphisme est la capacité, pour une fonction, d'être interprétée différemment en fonction de l'objet auquel elle s'applique.

On distingue

* Le polymorphisme statique : c'est la surcharge. Le sens de la fonction est déterminé statiquement par la signature des arguments à l'appel.
* Le polymorphisme dynamique : c'est le véritable polymorphisme. Le sens de la fonction est déterminé par le type effectif de l'objet à l'appel.

Le polymorphisme ne s'applique qu'aux méthodes d'instances.

Avantages:

* encourage l'abstraction,
* facilite l'extensibilité.

Méthodes virtuelles

Les *méthodes virtuelles* sont des fonctions membres réalisant le polymorphisme, c'est-à-dire le choix de la méthode en fonction du type des arguments à l'appel : un argument d'une classe dérivée utilise les méthodes de sa classe dérivée, et non de la classe du paramètre formel. En C++ (et en Java), le polymorphisme n'est réalisable que sur le "zéro-ieme" argument, c'est-à-dire sur l'objet appelant la méthode.

De plus, en C++ deux conditions doivent être réunies pour que le polymorphisme prenne effet (en Java, c'est le comportement par défaut)

1) La méthode doit être déclarée virtuelle, par le préfixe `virtual`
2) La méthode est appelée sur l'objet manipulé par pointeur ou par référence, pour qu'il n'y ait pas d'édition de lien statique.

Exemple

Déclaration de trois classes

```
class X {
public : int i; // i public pour que les classes dérivées puisse y accéder
  virtual void print();
  void show();
  X() { i=5;}
};

class XX : public X {
public :
  int j;
  virtual void print();
  void show();
  XX() { j=7;}
};

class XXX : public XX {
public :
  int k;
  void print();
  void show();
  XXX() { k=9;}
};
```

Chacune (sauf XXX) a une méthode `print` virtuelle, et une méthode `show` non virtuelle.
Définition des diverses méthodes. Noter : on ne répète pas le mot-clé `virtual` dans la définition.

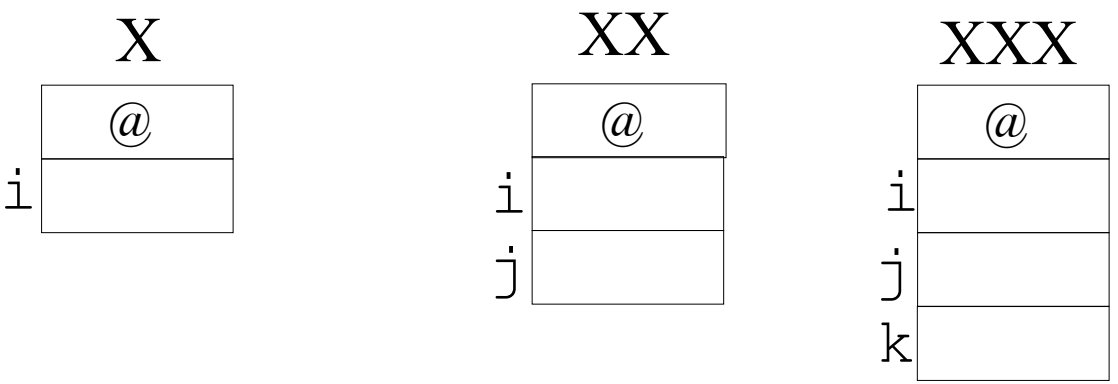
```
void X::print() { cout << "La classe est X " <<i <<endl; }
void X::show() {  cout << "La Classe est X " <<i <<endl; }

void XX::print(){ cout << "La classe est XX "<<i <<" "<<j<< endl; }
void XX::show() { cout << "La Classe est XX "<<i <<" "<<j<< endl; }

void XXX::print(){cout << "La classe est XXX "<<i <<" "<<j<<" "<<k<< endl;}
void XXX::show() {cout << "La Classe est XXX "<<i <<" "<<j<<" "<<k<< endl;}
```

La méthode utilisée dépend de la classe de l'objet, et la classe de l'objet est définie à la déclaration.

Forcer la conversion est possible, mais risqué car on accède à une zone mémoire non allouée.



Appel Direct

```
void main() {
  X z;
  XX zz;
  XXX zzz;

  z.print(); z.show();
  zz.print(); zz.show();
  zzz.print(); zzz.show();
  z = zz; z.print(); z.show(); // Conversion, ok
  zz = (XX) z; zz.print(); zz.show(); // danger
  zzz = (XXX) zz; zzz.print(); zzz.show(); // danger
}
```

```
La classe est X 5
La Classe est X 5
La classe est XX 5 7
La Classe est XX 5 7
La classe est XXX 5 7 9
La Classe est XXX 5 7 9
La classe est X 5
La Classe est X 5
La classe est XX 5 25720520
La Classe est XX 5 25720520
La classe est XXX 5 25720520 25720524
La Classe est XXX 5 25720520 25720524
```

Utilisation de pointeurs vers des objets.

Appel sur l'objet

```
void main() {
  X* a;  XX* aa;  XXX* aaa;
  a = new X; a -> print(); a -> show();
  aa = new XXX; aa -> print(); aa -> show();
  aaa = (XXX*) new X; aaa -> print(); aaa -> show();
}
```

Les résultats sont:

La classe est X 5
La Classe est X 5
La classe est XXX 5 7 9
La Classe est XX 5 7
La classe est X 5
La Classe est XXX 5 969289320 14

* La méthode virtuelle appliquée est celle de la classe de l'objet à sa création.
* La méthode non virtuelle est celle de la classe de la variable désignant l'objet.

Plus formellement:

Si une classe `Base` contient une fonction `f` spécifiée `virtual`, et si une classe `Der` dérivée contient une fonction `f` de même signature, l'appel de `f` sur un objet de classe `Der` invoque `Der::f` même si l'accès se fait à travers un pointeur ou une référence sur Base.

C'est ce qui permet le polymorphisme de nouvelles classes pouvant être créées après l'écriture du code d'appel de `f`, les nouvelles implémentations de `f` seront prise en compte.

Par contre:

Le polymorphisme (dynamique) ne s'applique pas aux paramètres de fonctions: la sélection en cas de surcharge est faite sur le type statique du pointeur (il n'y a pas d'argument `this`).

```
void afficher(Base* b) {...}
void affficher(Der* d) {...}
main() {
Base* x = new Der;
afficher(x); } // utilise la premiere fonction !
```

```
class Animal {
  protected:
    string nom;
  public:
    Animal(string n) : nom(n) {}
    virtual void show() { cout << nom << endl; }
};

class Ours: public Animal {
    string couleur;
  public:
    Ours(string n, string c) : Animal(n), couleur(c) {}
    void show() { cout << nom <<' '<< couleur << endl; }
};
```

Quel est le résultat de:

```
  Animal* o = new Ours("Panda","blanc");
  o->show();
```

Même résultat pour

```
  Ours b = Ours("Ours","brun");
  Animal& O = b;
  O.show();
```

On veut grouper, dans un tableau, des textes et des entiers (comme par exemple dans un tableur). Les données de base dérivent d'une classe commune:

```
class Item {
 public:
  virtual void show() = 0;//Virtuel pure => classe abstraite
};

class Texte: public Item {
 protected:
  string t;
 public:
  Texte(string t) : t(t) {}
  void show() {cout <<"Texte : " << t <<endl ; }
};

class Entier: public Item {
  int val;
 public:
  Entier(int val) : val(val) {}
  void show() { cout << "Entier : " << val << endl; }
};
```

La classe Item est abstraite, sa fonction show() est virtuelle pure, la classe Item ne peut être instanciée, elle correspond à un concept.

```
void main() {
  Item* donnee[10];
  donnee[0] = new Texte("Bonjour");
  donnee[1] = new Entier(2000);
  donnee[2] = new Texte("monde");

  for (int i = 0; i < 3; i++)
    donnee[i] -> show();
}
```

On peut réaliser le design pattern composite :

```
class CompositeItem : public Item {
 protected:
  vector<Item *> V;
 public:
  CompositeItem(Item *x) {V.insert(V.begin(), x);}
  void show() {
  vector<Item *>::iterator first = V.begin(), last;
  for (; first != last; ++first)
    (*first)->show();
    }
    // d'autre fonctions de gestion des éléments du composite
};
```

Virtuelle Pure ou Virtuelle par défaut

On aurait aussi pu définir

```
class Item {
 public:
  virtual void show(){};
};
```

Alors, la fonction show n'est plus virtuelle pure, la classe Item peut être instanciée, et la définition de show donne un comportement par défaut.

```
class Figure {
 public:
  virtual void draw() =0;  //pure
  virtual void paint(){};  //rien par defaut
};

class Rectangle : public Figure{
 public:
  void draw() {...};  //masque
  void paint(){...};  //masque aussi
};

class Segment : public Figure {
 public:
  void draw() {...};  //masque
  // ne masque pas paint
};
```

Un exemple: les expressions

Evaluation des expressions arithmétiques, sous le double aspect:

* Construction de l'arbre d'expression,
* Evaluation de cet arbre.

L'expression est lue sous forme postfixée. L'analyseur construit progressivement la pile des arbres des sous-expressions reconnues, et à la fin imprime la valeur de l'expression.

Voici la fonction principale

```
void main()
{
  Expression e;
  cout << "Donnez une expression postfixee\n";
  e = construction();
  cout << "Expression : ";
  e -> print();
  cout << "\nValeur : "<<e -> eval()<<endl;
}
```

un exemple d'exécution

Une expression est un concept abstrait : seules existent des sommes, produits etc, et des nombres. On définit donc les expressions comme classes abstraites.

Donnez une expression postfixee
3 4 + 2 - 2 / !
Je lis un entier 3
Pile : 3 . = 3
Je lis un entier 4
Pile : 4 3 . = 4
Je lis un char +
Pile : + 3 4 . = 7
Je lis un entier 2
Pile : 2 + 3 4 . = 2
Je lis un char -
Pile : - + 3 4 2 . = 5
Je lis un entier 2
Pile : 2 - + 3 4 2 . = 2
Je lis un char /
Pile : / - + 3 4 2 2 . = 2
Je lis un char !
Pile : ! / - + 3 4 2 2 . = 2
Expression : ! / - + 3 4 2 2
Valeur : 2

Voici la déclaration de la classe des expressions.

```
typedef class ExpressionR* Expression;
class ExpressionR {
public:
  virtual void print() =0;
  virtual int eval() =0;
};
```

Chaque opération arithmétique est représentée par une sous-classe. Il y en a 6:

Addition
Soustraction
Multiplication
Division
Factorielle
Simple

Les quatre premières sont ``binaires'', la Factorielle est ``unaire'', la dernière est zéroaire. Voici la déclaration des sous-classes (Une seule des classes d'opérateurs binaires est montrée).

```
class Soustraction: public ExpressionR {
    Expression gauche, droite;
  public:
    Soustraction(Expression, Expression);
    int eval();
    void print();
};

class Factorielle: public ExpressionR {
    Expression unique;
  public:
    Factorielle(Expression);
    int eval();
    void print ();
};

class Simple: public ExpressionR { //Zeroaire
    int valeur;
  public:
    Simple(int);
    void print();
    int eval();
};
```

Définition des méthodes des classes.

```
Soustraction::Soustraction(Expression a,Expression b) :
  gauche(a), droite(b) {}
void Soustraction::print() {
  cout << "- " ; gauche -> print(); droite -> print();
}
int Soustraction::eval() {
  return gauche -> eval() - droite -> eval();
}


Factorielle::Factorielle(Expression a): unique(a){}
void Factorielle::print() {
  cout << " ! " ; unique -> print();
}
int Factorielle::eval() {
  int n, v = unique -> eval();
  n = v;
  while (--v) n *= v;
  return n;
}


Simple::Simple(int n) : valeur(n) {}
void Simple::print() { cout << valeur << " " ;}
int Simple::eval() { return valeur;}
```

L'analyse syntaxique d'une expression utilise une fonction globale barbare `getlex(x,i)` qui retourne

* -1 à la fin de la lecture;
* 0 si le lexème lu est un entier;
* 1 si le lexème est un des caractères désignant un opérateur binaire ou la factorielle.

La construction de l'arbre est fonction du lexème retournée.

```
Expression construction() {
  Pile p; char x;
  Expression g, d; int i;
  for (;;) {
    switch (getlex(x,i)) {
    case -1:
      return p.pop();
    case 0:
      p.push(new Simple(i)); break;
    case 1:
      if (x == '!')
        p.push(apply(x,p.pop()));
      else {
        d = p.pop();
        g = p.pop();
        p.push(apply(x, g, d));
      }
    }
  }
  return NULL;
}
```

La pile est une pile d'expressions.

La fonction apply retourne un noeud approprié en fonction de l'opérateur.

```
Expression apply (char x, Expression a, Expression b=0)
{
  switch (x) {
    case '+':
      return new Addition(a,b);
    case '-':
      return new Soustraction(a,b);
    case '*':
      return new Multiplication(a,b);
    case '/':
      return new Division(a,b);
    case '!':
      return new Factorielle(a);
  }
  return NULL;
}
```

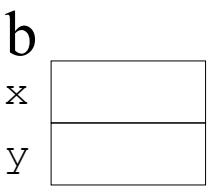
Note : D'autres problèmes se traitent de façon semblable. Il y a toute la famille de construction et d'évaluation d'arbres. Voir le design pattern ``Composite''

Table de fonctions virtuelles

Comportement « classique »

```
class Base { public:
    int x, y;
    int H(void);
  };
```

```
Base b;
```

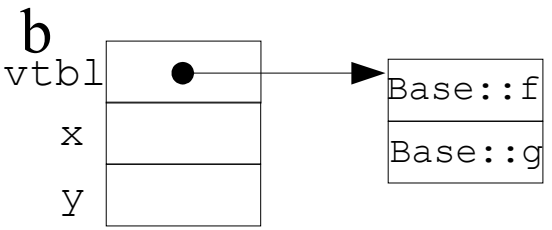


L'appel b.H() est converti en b.Base::H() c'est a dire un appel objet avec une adresse de méthode fixe.

Dans le cas de méthode Virtuelles

```
class Base {
  int x, y;
  int H(void);
  virtual int f(void);
  virtual int g(void);
  int truc();
};
```

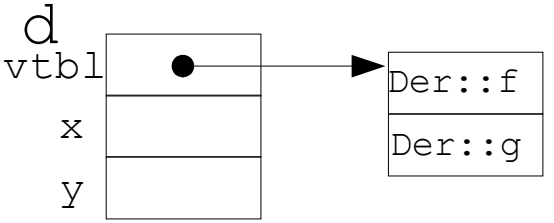
```
Base b;
```



L'appel `(&b)->g()` est remplacé par (&b)->vtbl[1]() à la compilation.
Le fait d'avoir une fonction virtuelle implique un pointeur supplémentaire par instance, qui pointe sur la table des fonctions virtuelles de l'instance. Ce qui défini d'une certaine mesure le type de l'instance .

Table des fonction virtuelle et héritage

```
class Der : Base {       Der d;
  public:
  int f(void);
  int g(void);
  int z;
};
```



L'appel `(&d)->g()` est remplacé par (&d)->vtbl[1]() à la compilation.
L'appel d.truc() est remplacé par d.Base::truc() à la compilation.

Soit maintenant

```
Base * p = new Der;
```

L'affectation au pointeur `p`

* restreint l'accès aux champs de Base, i.e. `p -> z` est une erreur;
* ne modifie pas le contenu des champs de l'objet pointé: `p -> g()` vaut `p -> vtbl[1] = Dg()`.

Par contre une conversion `b = d` remplace la table de Der par la table de Base.

On appelle ligature dynamique ou résolution ou typage dynamique le calcul de l'adresse de la fonction appelée en fonction du type de l'objet appelant, calcul fait à l'exécution (lecture de la table) .