GL / C++
Chapitre 7

Lien Dynamique
Méthodes Virtuelles

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

1. Pointeur sur un objet d'une classe dérivée

- Considérons les classes écrites précédemment :

```
Personne
Etudiant     // dérive de personne
Salarie      // dérive de personne
```

- Le langage C++ permet d'écrire les déclarations suivantes :

```
Personne * p1 =
        new Personne("dupond","10 rue de l'isère");
Etudiant * p2 =
        new Etudiant("durand", "15 rue de l'isere");
Personne * p3 =
        new Salarie("dumoulin", "21 rue jean jaures",
                    "schneider", 12000.00);
```

- Dans p3, déclaré pointeur sur Personne, on peut mettre l'adresse d'un objet de type Salarie.
- Ceci est possible car la classe Salarie est dérivée de la classe Personne (autrement dit : un Salarié est une Personne particulière).

Remarque

- Il serait incorrect d'écrire :

```
Personne p =
  Salarie("dumoulin", "21 rue jean jaures",
          "schneider",12000.00);
```

- ... car dans un objet de la classe Personne on ne peut pas stocker un objet de la classe salarie (un objet salarie occupe plus de place en mémoire qu'un objet de la classe personne)

2. Lien statique - lien dynamique

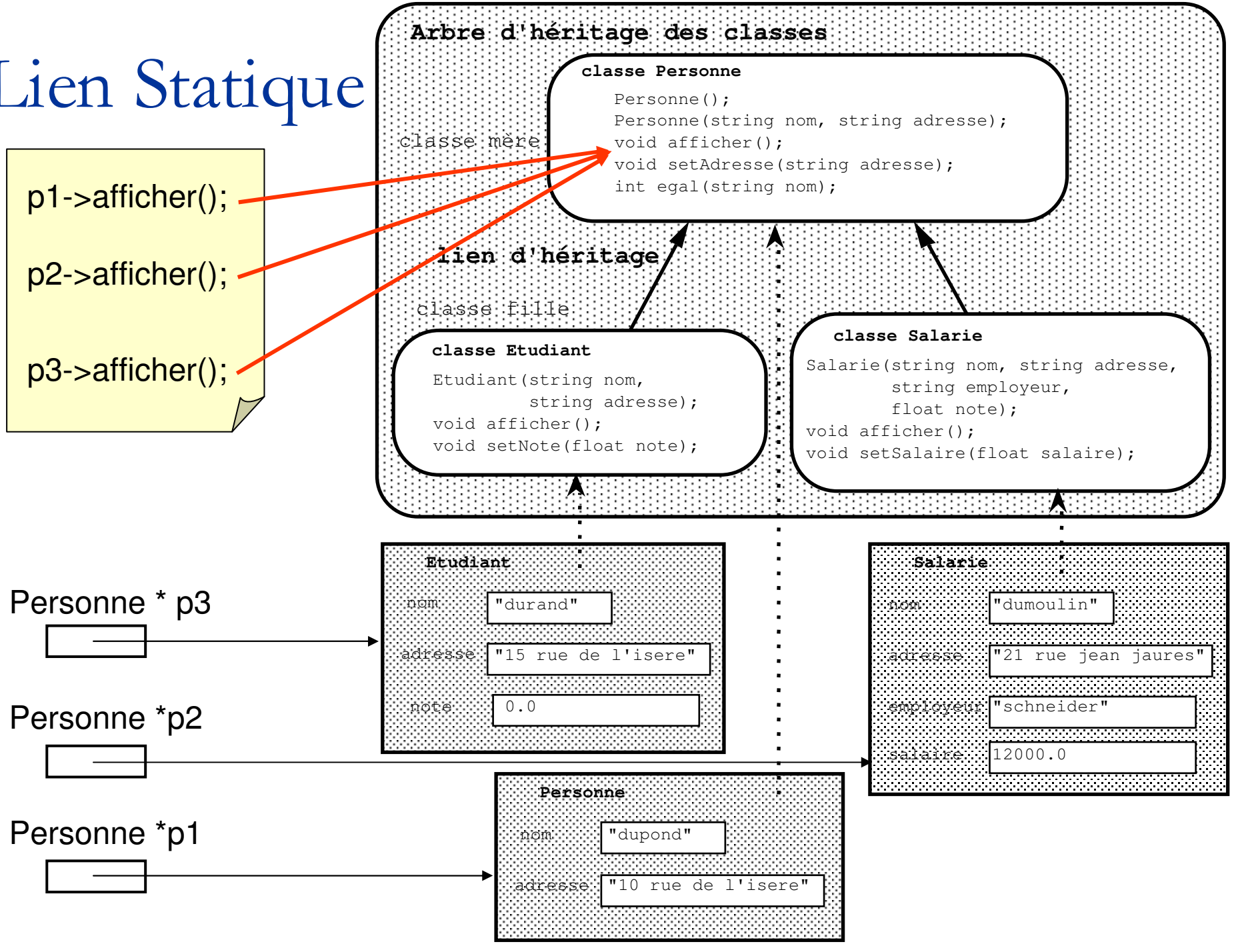
- Considérons le code suivant :

```
int main()
{
  Personne * p1 =
    new Personne("dupond", "10 rue de l'isere");
  Personne * p2 =
    new Salarie("dumoulin", "21 rue jean jaures",
                "schneider", 12000.00);
  Personne * p3 =
     new Etudiant("durand", "15 rue de l'isere");

  p1->afficher(); // envoi de message (1)
  p2->afficher(); // envoi de message (2)
  p3->afficher(); // envoi de message (3)

  return 0;
}
```

Lien Statique



2. Lien statique - lien dynamique

- L'envoie de message (1) appellera **`Personne::`**`afficher` car
  - `p1` est déclaré pointeur sur un objet de classe **`Personne`**
  - le compilateur sait donc qu'il faut appeler cette méthode.
- Pour la même raison (`p2` et `p3` de type pointeur sur `personne`) :
  - (2) et (3) appelleront **`Personne::`**`afficher`.
- Or la méthode appelée devrait logiquement être :
  - **`Salarie::`**`afficher` pour l'envoie de message (2)
  - **`Etudiant::`**`afficher` pour l'envoie de message (3)

2. Lien statique - lien dynamique

- **Au moment de la compilation**, le compilateur :
  - ne connaît pas la classe de l'objet qui, **lors de l'exécution,** sera pointé par **`p2`**
  - ne peut donc pas appeler la méthode qui convient pour afficher cet objet.
- Le compilateur doit mettre en place un mécanisme qui :
  - **au moment** de l'exécution (dynamiquement)
  - appelle la méthode de la classe de l'objet pointé par le pointeur

2. Lien statique - lien dynamique

- Le Mécanisme appelé **lien dynamique** est un lien :
  - qui détermine la méthode appelée à l'exécution du programme
  - qui n'appelle pas toujours la même méthode (pas systématiquement celle de la classe du pointeur)

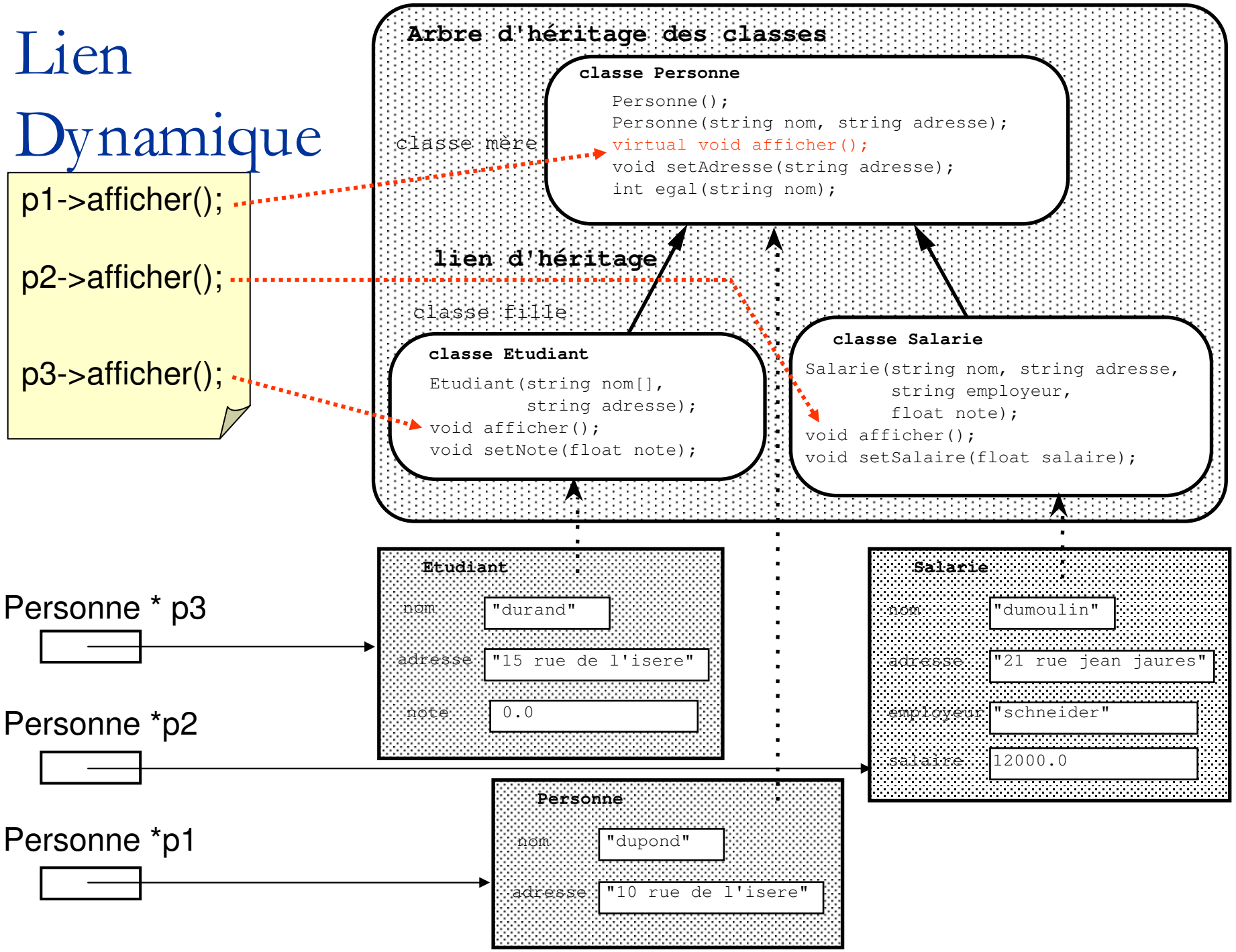
- on parle de **lien statique** lorsque :
  - le compilateur appelle la méthode de la classe du pointeur
  - C'est un lien qui appelle toujours la même méthode

Lien Dynamique

p1->afficher();

p2->afficher();

p3->afficher();



3. Méthode virtuelle

- Considérons l'envoi de message :

```
Personne * p;
p = new Etudiant("durand", "15 rue de l'isere");
// autres instructions
p->afficher();  // doit appeler Etudiant::afficher
                // lien dynamique
```

- Pour que le lien `->afficher()` soit dynamique, Il faut que dans la classe personne, la méthode `afficher` ait été déclarée **virtuelle** (**virtual**) dans la classe Personne

4. Classe Personne avec méthode virtuelle

```
class Personne {

protected:
      string nom;
      string adresse;
public:
      Personne();
      Personne(string nom, string adresse);

      virtual void afficher();
      void setAdresse(string adresse);
      int  egal(string nom);
};
```

4. Classe `Personne` avec méthode virtuelle

- Le compilateur mettra alors en place le mécanisme de lien dynamique pour tous les envois de message tels que :

```
Personne * P;
p=new Etudiant("durand", "15 rue de l'isere");
p->afficher(); // Etudiant::afficher() est appelé
```

- **Remarque**
  - Supposons que l'on déclare la classe `SalarieAuMois` dérivée de `Salarie`
  - Le compilateur ne mettra pas en place le mécanisme de lien dynamique pour les envois de message tel que :

```
Salarie * p;
p=new SalarieAuMois("durand", "15 rue de
                             l'isere", ....);
p->afficher(); // Salarie::afficher() est appelé
```

  - Pour cela il faudrait que la méthode `afficher()` de la classe Salarie ait aussi été déclarée virtuelle

5. Quand une méthode doit-elle être déclarée virtuelle ?

- Une méthode **`maMethode()`** d'une classe **`MaClasse`** doit être déclarée virtuelle :
  - si on sait que **`maMethode()`** sera redéfinie dans des classes dérivées de **`MaClasse`**
  - et si on sait que des objets de la classe **`MaClasse`** ou de classes dérivées recevront des envois de message **`maMethode()`** en étant manipulés *via* des pointeurs :

```
MaClasse * p;            // ou  MaClasse * tab[10];
// affectations de adr_obj ou de tab[i]
p->maMethode();          // ou tab[i]->maMethode();
```

6. Remarques

- Dans beaucoup de langages orientés objets, les méthodes sont toutes implicitement virtuelles
- C'est pour des raisons d'efficacité que C++ oblige le programmeur à spécifier explicitement les méthodes virtuelles qui seront appelées avec un lien dynamique
- En effet un lien statique est plus efficace qu'un lien dynamique
- Avec le compilateur **g++**, la directive de compilation **-fall_virtual** permet de demander au compilateur de considérer toutes les méthodes comme virtuelles… cela se fait bien sûr au détriment de l'efficacité !

GL / C++
Chapitre 8

Polymorphisme

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

1. Principe

- Considérons (encore !) les classes écrites précédemment :

```
Personne
Etudiant      // dérivée de Personne
Salarie       // dérivée de Personne
```

- On peut écrire une procédure qui manipule un tableau de pointeurs sur des Personnes dont les éléments peuvent pointer :
  - soit vers des Personnes
  - soit vers des Salariés
  - soit vers des Etudiants.

Exemple

```
int main()
{
        Personne * tab[10]; // tableau de pointeurs
        int nbElem = 0;     // nombre d'éléments présents
                            // dans le tableau

        tab[nbElem++] = new Personne("dupond",
                              "10rue de l'isere");
        tab[nbElem++] = new Etudiant("durand",
                              "15rue del'isere");;
        tab[nbElem++] = new Salarie("dumoulin",
                              "21 rue jean jaures",
                              "schneider",12000.00);

        for (int i=0; i<nbElem; i++)
              tab[i]->afficher(); // tab[i] pointeur

        return 0;
}
```

Remarques

- Etudiant et Salarie étant dérivées de Personne, il est correct de ranger dans des éléments du tableau des pointeurs sur des objets de type etudiant et salarie.
- la méthode **`Personne::afficher`** étant virtuelle, `tab[i]->afficher()` appellera :
  - la méthode **`afficher`** particulière de l'objet `*tab[i]`
  - et non la méthode de la classe du pointeur (`Personne`)

2. Structure de donnée polymorphe

- Le tableau `tab`
  - peut donc contenir des pointeurs sur des instances de toute classe dérivée de personne
  - et la boucle d'affichage appellera la méthode d'affichage propre à chaque objet pointé par le tableau.
- Une telle structure de données est dite **polymorphe**, pour exprimer le fait qu'elle peut contenir des objets :
  - de classes différentes
  - c'est-à-dire de formes différentes (poly-morphe).

2. Structure de donnée polymorphe

- Le **polymorphisme** est la possibilité de définir des structures de données polymorphes
- Grâce à l'héritage d'une part et au lien dynamique d'autre part, les langages orientés objets permettent de définir des structures de données polymorphes :
  - **Listes polymorphes** : listes dont les informations sont des pointeurs sur des objets hétérogènes
  - **tableaux polymorphe** : tableaux dont les éléments sont des pointeurs sur des objets hétérogènes

GL / C++
Chapitre 9

Généricité
Patrons (*templates*)

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

Généralités

- **1ère façon « implicite » de réaliser la généricité : celle déjà évoquée**
  - Nous avons réalisé des structures de données relativement indépendantes de la classe de leurs données (c'est-à-dire de leur type) avec les structures de données polymorphes évoqués précédemment.
  - On peut aussi écrire des fonctions travaillant sur de nombreux types grâce à la surcharge.
- **2ème façon : utilisation de patrons de fonction ou de classe**
  - Les paramètres **template** permettent de paramétrer la définition des fonctions et des classes.
  - Un paramètre **template** est soit :
    - un **type (générique)** : on parle de **paramètre de type**
    - un paramètre « ordinaire » : on parle de **paramètre expression**
  - Les fonctions et les classes ainsi paramétrées sont appelées respectivement **patron de fonction** et **patron de classe**

Patron de fonction

- Un **patron de *fonction*** est donc une fonction générique :
  - dont certains paramètres sont de **types génériques**. Un type générique est appelé **paramètre de type** du patron de fonction.
  - dont les éventuels autres paramètres sont des paramètres « ordinaires ». Ces paramètres sont des **paramètres expression** du patron de fonction
- Chaque **paramètre de type** doit apparaître au moins une fois dans l'en-tête du patron
- Exemple : patron "minimum de 2 valeurs a et b de type T"

```
template <class T> T min (T a, T b) {
   if (a<b) return a; else return b;
}
```

  - T est un "paramètre de type" du patron de fonction "min"
  - a et b sont des "paramètres expression" du patron "min"

Patron de classe

- Un **patron de classe** est une classe générique qui fait intervenir, dans la définition de ses membres (attributs ou méthodes) :
  - Des types génériques : un tel type est donc considéré comme un **paramètre de type** du patron de classe
  - Des **paramètres expressions** : ce sont des paramètres « ordinaires » (int, float, ..) qui permettent de paramétrer la définition du patron.
- Exemple : patron "tableau de n éléments de type T"

```
template <class T, int n=10> class Tableau {
   T tab[n];
   public:
      Tableau();
      void ajouterElement(T e);
      ...
};
```

  - **T** est un paramètre de type du patron de classe "Tableau"
  - **n** est un paramètre expression du patron de classe " Tableau "
  - on peut donner une valeur par défaut aux paramètres "*templates*"

Généralités (suite)

- la génération du code (l'*instanciation des patrons*)
  - les paramètres génériques sont remplacés par de vrais types
  - les paramètres expressions prennent leur valeur
- L'instanciation du patron a lieu lorsque l'on utilise le patron de fonction ou de classe pour la 1ère fois.
- Les types réels à utiliser à la place des types génériques sont déterminés par le compilateur lors de cette première utilisation :
  - soit implicitement à partir du contexte d'utilisation du patron
  - soit explicitement par les paramètres donnés par le programmeur lors de l'utilisation du patron

C++ permet donc de définir :

- Des fonctions et des classes **génériques** appelées respectivement patrons de fonction et patrons de classe

- Les **paramètres de généricité** sont essentiellement des **types de données**
  - types de base (int, float, ...)
  - et/ou classes

Exemple de procédure générique

- Définition d'une procédure patron permettant de permuter le contenu de 2 variables d'un type quelconque (fichier `permut.h`)

```
// paramètre de généricité
// T représente le type des variables permutées
// Les caractères <> sont obligatoires

template <class T> void permut(T & e1, T & e2) {

    T e = e1;
    e1 = e2;
    e2 = e;
}
```

Exemple d'utilisation (instanciation)

- Utilisation pour permuter des Personnes, des entiers des float

```
#include "Personne.h"
#include "permut.h"

int main() {

  Personne p1("dupond", "10 rue des fleurs");
  Personne p2("durand", "15 rue des cactus");
  permut(p1, p2); // instanciation implicite
                  // permut(Personne&, Personne&)
  int i1=3, i2=5;
  permut(i1, i2); // instanciation implicite
                  // permut(int&, int&)

  float f1=2.0, f2=5.0;
  permut<float>(f1,f2); // instanciation explicite
                        // permut(float&, float&)
  return 0;
}
```

Classe Tableau générique : spécification

```
// paramètre de généricité = type des éléments du tableau

template <class T> class Tableau {

protected:
  int nbMax;       // nombre max d'éléments de tab
  T * tab;         // tableau dynamique d'éléments de type T
  short nbElem;    // nombres d'éléments présents dans tab

public:
  Tableau(int nbMax);        // construit un tableau vide
  int nbElements();          // retourne le nbre d'éléments de tab
  void ajouter(T e);         // ajoute e au tableau
  void supprimer(int i);     // supprime le ième élément du tableau
  T operator[](int i);       // retourne l'élément i s'il existe

};
```

Implémentation : Tableau.cc

```
template <class T> Tableau<T>::Tableau(int nbMax) {
        nbElem = 0;
        this->nbMax = nbMax;
        tab = new T [nbMax];
}

template <class T> int Tableau<T>::nbElements() {
        return nbElem;
}
```

Implémentation : tableau.cc (suite)

```
template <class T> void Tableau<T>::ajouter(T e) {
        if ( nbElem < nbMax ) tab[nbElem++]=e;
}

template <class T> void Tableau<T>::supprimer(short i) {
        if ( i<0 || i >= nbElem )  return; // indice incorrect
        nbElem--; // nouvelle taille
        if ( i != nbElem-1 )
          tab[i] = tab[nbElem-1];
          // suppression de tab[i] par permutation avec le
          // dernier élément (méthode grossière qui ne
          // conserve pas l'ordre des éléments du tableau !)
}

template <class T> T Tableau<T>::operator[](short i) {
        if ( i<0 || i >= nbElem ) return NULL;
        else return tab[i];
}
```

Remarque 1

- Chaque méthode est précédée de la déclaration suivante qui rappelle les paramètres de généricité :

```
template <class T>
```

- Le nom de chaque méthode est précédé du nom de la classe qui est paramétré, entre crochets, par le paramètre de généricité

```
void Tableau<T>::supprimer(short i)
```

Remarque 2

- Dans un programme qui utilise un patron, **il faut inclure l'implémentation de ce patron.** Deux solutions :
  - Comme d'habitude, on écrit la spécification et l'implémentation dans 2 fichiers distincts, **`MonTemplate.h`** et **`MonTemplate.cc`**, mais l'implémentation devra alors être incluse dans la spécification :

MonTemplate.h

```
#ifndef MONTEMPLATE_H
#define MONTEMPLATE_H
// code du .h
#include "MonTemplate.cc"
#endif
```

MonTemplate.cc

```
#ifdef MONTEMPLATE_H
// code du .cc
#endif
```

  - On peut aussi écrire la spécification et l'implémentation dans un seul fichier : **`MonTemplate.h`**
- Dans les deux cas, on inclura donc "MonTemplate.h" lorsque l'on veut utiliser le template.

Exemple d'utilisation

```
#include "personne.h"
#include "tableau.h"

int main() {
  tableau <int> t1(10);
  // tableau d'au plus 10 entiers
  t1.ajouter(1); t1.ajouter(2); t1.ajouter(3);
  for (i=0; i<t1.nb_elements(); i++)
    cout<< t1[i] << endl;

  tableau <personne *> t2(10);
  // tableau d'au plus 10 pointeurs sur des personnes
  t2.ajouter(new personne("dupond", "10 rue des fleurs"));
  t2.ajouter(new personne("durand", "25 rue des fleurs"));
  for (i=0; i<t2.nb_elements(); i++)
    t2[i]->afficher();

  return 0;
}
```

Exercices 1 & 2

1.1. Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque (le résultat sera du même type)

1.2. Ecrire un petit programme utilisant ce patron

2.1. Créer un patron de classes nommé `PointColore` tel que chaque classe instanciée permette de manipuler des points colorés (deux coordonnées et une couleur) pour lesquels on puisse choisir à la fois le type de coordonnées et celui de la couleur.

La classe comportera deux méthodes : un constructeur à 3 arguments et une fonction **affiche** affichant les coordonnées et la couleur d'un point coloré

2.2. Ecrire un petit programme utilisant cette classe avec différentes instanciations du patron

2.3. Dans quelle condition peut-on instancier une classe patron `PointColore` pour des paramètres de type classe ?

Exercice 3

- On dispose du patron suivant :

```
template <class T> class Point {

    T x, y ;     // coordonnees

  public :

    Point (T abs, T ord) {
      x = abs ; y = ord ;
    }

    void affiche () {
      cout << "Point - coordonnees " << x << " " << y;
    }
} ;
```

- On souhaite créer un patron de classes cercle permettant de manipuler des cercles définis par leur centre (un point) et un rayon.
  - Le faire par héritage (un cercle est un point qui possède un rayon)
  - Le faire par composition d'objets membres