Cours 4 : Le polymorphisme

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

- La notion de polymorphisme
- Polymorphisme des fonctions
- Polymorphisme dans l'héritage
- Compatibilité des objets et conversion
- Mot clé virtual : héritage virtuel, méthodes virtuelles et méthodes virtuelles pures
- Abstraction dans l'héritage

Principe (1)

Définition : **Polymorphisme**

- = Plusieurs formes
- Pour les méthodes
- Pour les objets dans l'héritage

On parle alors de polymorphisme dans plusieurs contextes différents :

- Polymorphisme des fonctions
- Polymorphisme dans l'héritage

Principe (2)

Polymorphisme des fonctions :

- La possibilité de définir plusieurs fonctions avec : le même nom et des paramètres de type et de nombre différent

Polymorphisme dans l'héritage :

- La capacité d'appeler une méthode en fonction du type de l'objet appelant (sa hiérarchie dans l'héritage)

- La possibilité de traiter plusieurs formes d'une classe :
  - Le cas de plusieurs classes héritant d'une classe abstraite

Polymorphisme des fonctions (1)

- Contrairement au langage C, le langage C++ intègre le polymorphisme des fonctions :

Exemple : `fct(int a), fct(int a, int b), fct(int a, string b) …`

- Chaque fonction est identifiée par une « clef » basée sur son nom et le type et l'ordre des paramètres qu'elle accepte

Remarque :

Le polymorphisme de fonctions peut aussi désigner le fait de pouvoir définir des fonctions de même nom et paramètres dans des classes différentes

Polymorphisme des fonctions (2)

```
class Nombre {
public:
        int valeur;
        Nombre() { } // Const par défault
        Nombre(int val){ valeur = val;}
        void affiche(int val)
                { cout << val << endl; }
        void affiche(string val)
                { cout << val << endl; }
        void affiche(Nombre *objet)
                { cout << objet->valeur <<
                endl; }
};
```

```
#include <string>
int main() {
        Nombre A;
        Nombre * B = new Nombre(7);
        A.affiche("3");
        A.affiche(5);
        A.affiche(B);
        return 0;
}
// Le programme affiche : 3, 5 et 7
```

Polymorphisme dans l'héritage

Compatibilité des objets :

- Tout objet d'une classe dérivée peut être traité et utilisé comme un objet de sa classe de base

Méthodes virtuelles :

- Méthodes virtuelles pour gérer la redéfinition des fonctions (surcharge)
- Méthodes virtuelles pures pour créer des classes abstraites

Compatibilité des objets et conversion (1)

- Une conversion implicite d'un objet d'une classe dérivée B en un objet de la classe de base A est possible si l'héritage est public

- L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres supplémentaires de la classe dérivée

Principe de conversion :

Classe fille ➔ Classe mère Possible

Classe mère ➔ Classe fille Impossible

Compatibilité des objets et conversion (2)

Exemple :

```
class A {};
class B : public A {};
int main() {
        A a;
        B b;   // B est une classe dérivée de A
        a = b; // correct, seule la partie A de b est copiée dans a
        b = a; // incorrect, impossible de mettre A dans B. Il manque des champs
        A* pa = &a;  // correct bien sur
        pa = &b;     // correct aussi
        B* pb = &a;  // erreur
        return 0;
}
```

Mot clé virtual

Héritage multiple :

- `class B : virtual public A {};`
- `class C : virtual public A {};`
- `class D : public B : public C {};`

➔ Permet de ne pas dupliquer les attributs et méthodes de A dans D

Méthodes virtuelles :

- `virtual void fct ();`

➔ Permet d'appeler la méthode `fct ()` en fonction de la nature de l'objet

Méthodes virtuelles pures :

- `virtual void fct () =0;`

➔ Permet de créer une classe abstraite

Méthodes virtuelles (1)

- Le polymorphisme consiste en les différentes formes que peut prendre l'implémentation d'une méthode définie plusieurs fois (surchargée)
- Cette méthode change en fonction du type d'objet qui l'appelle
- En C++, on parle alors de méthode virtuelle : méthode à laquelle on rajoute le mot-clé virtual dans la définition

Syntaxe :

```
virtual void fct ();
```

- Considérons l'exemple suivant qui se base sur les Carres et Rectangles

Méthodes virtuelles (2)

```
class Rectangle {
public: Rectangle(int lon, int lar);
        int longueur;
        int largeur;
        int calculer_perimetre()
        {return 2 * longueur + 2 *
        largeur; }
};
class Carre : public Rectangle {
public: Carre(int cote);
        int calculer_perimetre()
        {return 4 * largeur; }
};
```

```
int main() {
        Rectangle R1(10, 10);
        Carre R2(10);
        Rectangle * ptr;
        ptr = &R2;
        cout << ptr->calculer_Perimetre()
        << endl;
}
// Ici on a une liaison statique, le
// compilateur utilise la fonction
// calculer_Perimetre() de Rectangle
```

Méthodes virtuelles (3)

- Dans le cas d'une liaison statique (décidée par le compilateur) la seule fonction qui sera appelée est `Rectangle::calculer_Perimetre()`

- Ce cas a peu d'intérêt, car nous avons défini une fonction `Carre::calculer_Perimetre()`

Problème :

On aurait souhaité exécuter `Carre::calculer_Perimetre()` dans le cas où `ptr` pointe sur une instance de type Carre

Méthodes virtuelles (4)

Solution :

Rendre la fonction `calculer_Perimetre()` virtuelle dans les classes Rectangle et Carre

- En demandant que la fonction `calculer_Perimetre()` soit définie virtuelle, on impose une liaison dynamique

- La fonction à appeler sera alors déterminée à l'exécution en fonction du type de l'objet

Méthodes virtuelles (5)

```
class Rectangle {
public: Rectangle(int lon, int lar);
        int longueur;
        int largeur;
        virtual int calculer_perimetre()
        {return 2 * longueur + 2 *
        largeur; }
};
class Carre : public Rectangle {
public: Carre(int cote);
        virtual int calculer_perimetre()
        {return 4 * largeur; }
};
```

```
int main() {
        Rectangle R1(10, 10);
        Carre R2(10);
        Rectangle * ptr;
        ptr = &R2;
        cout << ptr->calculer_Perimetre()
        << endl;
}
// Ici on a une liaison dynamique, le
// compilateur utilise la fonction
// calculer_Perimetre() de Carre
```

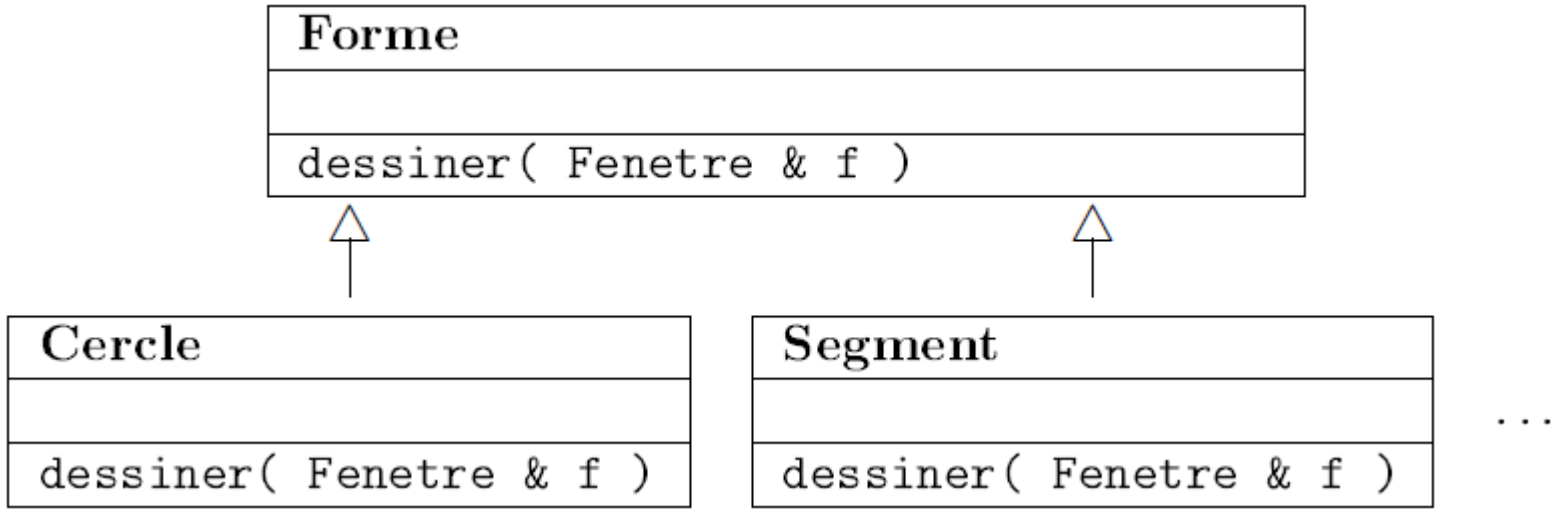
Méthodes virtuelles pures (1)

- Une méthode virtuelle pure est une méthode qui est déclarée mais non définie dans une classe
- Elle est définie dans une des classes dérivées de cette classe
- Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « =0 » :
- Syntaxe : `virtual void fct () =0;`
- L'information « =0 » signifie ici simplement qu'il n'y a pas d'instance de cette méthode dans cette classe

Méthodes virtuelles pures (2)

Exemple : Un programme manipule différentes formes géométriques :

- Segment de droite
- Carre
- Rectangle
- Cercle
- …



Objectif : On souhaite surcharger la méthode dessiner dans l'ensemble des formes géométrique

Méthodes virtuelles pures (3)

```
class Forme {
      virtual void dessiner(Fenetre & f) = 0;
      // Notez le '= 0' transforme la classe 'Forme' en classe abstraite
};
class Segment : public Forme {
      virtual void dessiner(Fenetre & f) {/*…*/}
      // La classe Segment contient sa propre définition de dessiner
};
class Cercle : public Forme {
      virtual void dessiner(Fenetre & f) {/*…*/}
      // La classe Cercle contient sa propre définition de dessiner
};
```

Abstraction dans l'héritage (1)

Définition : Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure

- On ne peut pas créer d'instance d'une classe abstraite
- Une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction

Objectif :

- Les méthodes virtuelles pures servent de cadre générique pour les méthodes virtuelles des classes dérivées
- Ceci permet de garantir une bonne homogénéité de l'architecture des classes

Abstraction dans l'héritage (2)

Remarque :

Si une classe hérite d'une classe abstraite, mais ne définit pas de corps à une méthode virtuelle pure héritée :

➔ Cette nouvelle classe est toujours une classe abstraite, non instanciable

Exemple :

```
class A {
public:
        virtual void fct() = 0;
};
class B : public A {};
```

```
int main() {
        A a;   // incorrect, A abstraite
        B b;   // incorrect, B abstraite
}              // tant que fct() non défini
```