Liaison dynamique et méthodes virtuelles

Pour illustrer l'intérêt de la liaison dynamique[6], considérons l'exemple suivant :

```
class polygone
{
  ...
  void affiche();
  ...
};
class rectangle : public polygone
{
  ...
  void affiche();
  ...
};
```

que l'on utilise dans un programme principal de cette manière :

```
polygone my_polygone;
rectangle my_rectangle;
polygone *ptr_polygone1 = &my_polygone;
polygone *ptr_polygone2 = &my_rectangle;
```

L'instruction `ptr_polygone1->affiche();` appelle la méthode `affiche` de la classe `polygone`. Or, si nous exécutons l'instruction `ptr_polygone2->affiche();` cette dernière fait également appel à la méthode `affiche` de la classe `polygone` et non à celle définie pour la classe `rectangle`. Ce comportement est du au choix de la méthode qui a lieu lors de la compilation du programme : `ptr_polygone2` étant un pointeur de type `polygone`, le compilateur lui a attribué l'ensemble des méthodes de la classe `polygone` et non celle de la classe `rectangle`. On parle alors de "ligature statique".
Pour pallier cette limitation et préserver les liaisons entre classes et méthodes, le C++ propose le mécanisme de fonctions virtuelles. Il suffit de déclarer "virtuelle" (mot clé `virtual`) la méthode `affiche` de la classe mère. La classe `polygone` de notre exemple devient :

```
class Polygone
{
  ...
  virtual void affiche();
```

```
  ...
};
```

Cette instruction indique au compilateur que les éventuels appels de la fonction `affiche` doivent utiliser une liaison dynamique et non plus statique. Autrement dit, lorsque le compilateur rencontrera un appel tel que :

```
ptr_polygone2->affiche();
```

il ne décidera pas de la procédure à appeler. Il se contentera de mettre en place un dispositif permettant de n'effectuer le choix de la méthode qu'au moment de l'exécution de cette instruction, le choix étant finalement basé sur le type exact de l'objet faisant l'appel.

Ce processus de virtualisation permet ainsi de redéfinir des méthodes[7] suivant la finalité de la classe fille. D'autre part, il s'inscrit parfaitement dans l'esprit de spécialisation inhérent à la notion d'héritage. La méthode `affiche` redéfinie dans la classe fille `rectangle` permettra, par exemple, d'afficher la longeur et la largeur du rectangle ou toutes autres informations propres à la classe `rectangle`.
Dans l'exemple précédent, l'instruction

```
polygone *ptr_polygone2 = &my_rectangle;
```

demeure valable du fait de conversions standards automatiquement définies entre une classe de base et ses classes dérivées. Ainsi, des conversions implicites permettent de :

- convertir un objet de type `rectangle` en un objet de type `polygone`,
- convertir un pointeur sur un objet de type `rectangle` vers un pointeur sur un objet de type `polygone`,
- convertir une référence sur un objet de type `rectangle` vers une référence sur un objet de type `polygone`.

Ces conversions ne présentent aucun risque particulier puisqu'un objet de type `rectangle` est avant tout un objet de type `polygone`. Dans le premier cas, c'est une conversion d'objet qui est effectuée : seuls les attributs définis dans `polygone` sont pris en compte, ceux supplémentaires définis dans `rectangle` ne sont pas pris en considération. Dans les deux autres cas (pointeurs et références), c'est uniquement une conversion de type qui est réalisée : l'objet en lui-même n'est pas affecté. Ce qui implique qu'un pointeur de type `polygone` puisse pointer vers un objet de type `rectangle` [8], l'inverse n'étant pas vrai.
L'instruction `rectangle *ptr_rectangle = &my_polygone` provoquera une erreur de compilation.