Chapitre 5

L'héritage

Une **classe** peut être **dérivée** d'une autre (cette dernière étant alors dite **classe de base** de la classe dérivée), ou même de plusieurs, en ce sens qu'on ajoute des attributs supplémentaires et/ou des méthodes supplémentaires. Plutôt que de redéfinir entièrement une telle classe, on peut indiquer qu'elle **hérite** des attributs et des méthodes de la première et ne porter toute son attention qu'aux éléments vraiment nouveaux. C'est la technique de l'**héritage**, que nous allons étudier dans ce chapitre.

On parle d'**héritage simple** lorsque la classe hérite d'une seule classe et d'**héritage multiple** lorsqu'elle hérite de plusieurs classes.

5.1 Héritage simple

Commençons par un exemple d'héritage simple.

Le problème.- Considérons le problème suivant de programmation. On tient à jour dans un établissement scolaire un répertoire des personnes le fréquentant. On distinguera traditionnellement les élèves, les enseignants et le personnel non enseignant, catégories pour lesquelles les renseignements conservés sont très différents.

On considèrera une **superclasse** (c'est-à-dire une classe dérivée d'aucune autre classe) `personne` avec les types de renseignements communs à chacune des trois *classes dérivées* de cette classe `eleve`, `enseignant` et `personnel` (sous-entendu non enseignant). Pour simplifier, nous ne nous occuperons ici que des classes `personne` et `eleve`.

Les seuls renseignements (et donc attributs) concernant une personne seront son nom (`nom`) et son numéro de téléphone (`tel`), nécessaires pour la contacter rapidement en cas de besoin. Les méthodes associées seront l'initialisation (`init()`) et l'affichage (`affiche()`) de ces renseignements.

Le seul renseignement supplémentaire que nous prendrons en compte ici pour un élève sera sa promotion (`promo`). Nous aurons besoin de méthodes pour initialiser et afficher ce renseignement.

5.1.1 Définition des classes dérivées

Introduction.- Nous allons voir comment définir une classe dérivée d'une autre classe. Commençons cependant par revoir comment nous aurions fait sans notion d'héritage.

Définition des classes sans héritage.- Bien entendu nous pouvons faire face au problème choisi à titre d'exemple sans utiliser la notion d'héritage de la façon suivante :

```
class personne
   {
   char nomM[30];
   char tel[20];
   public:
   void init(char *name, char *phone);
   void affiche(void);
   };

class eleve
   {
   char nom[30];
   char tel[20];
   char promo[10];
   public:
   void init(char *name, char *phone);
   void init_p(char *classe);
   void affiche(void);
   void affiche_p(void);
   };
```

en laissant le soin au lecteur, pour l'instant, de définir les fonctions membres.

Mise en place avec héritage.- La déclaration des premiers attributs et la définition des méthodes communes à chacune des trois classes étant les mêmes, on peut espérer gagner du code (et donc

de la place) et ne s'intéresser qu'aux choses importantes en utilisant la notion d'héritage. On a alors les déclarations suivantes.

```
class personne
   {
   protected:
   char nom[30];
   char tel[20];
   public:
   void init(char *name, char *phone);
   void affiche(void);
   };

class eleve: public personne
   {
   protected:
   char promo[10];
   public:
   void init_p(void);
   void affiche_p(void);
   };
```

Mises en place.- Lorsqu'on veut qu'une classe `B` soit dérivée d'une classe `A`, on l'indique dans l'en-tête de la définition de la classe `B` de la façon suivante :

```
class B : public A
```

la chaîne de caractères ‘ `: public`’ indiquant que les membres publics (et protégés) de la classe `A` sont aussi des membres de la classe `B`.

Membres publics, privés et protégés.- On aura remarqué que l'héritage nous invite à considérer un nouveau genre de membres : les **membres protégés**. Pour une classe elle-même un membre protégé se comporte de la même façon qu'un membre privé. C'est lors de l'héritage que se fait la différence :

- un membre public de la classe de base est un membre public de la classe dérivée ;
- un membre protégé de la classe de base est un membre protégé de la classe dérivée ;
- un membre privé de la classe de base n'est pas un membre de la classe dérivée ou, plus exactement on ne peut pas y avoir accès dans les fonctions de la classe dérivée.

5.1.2 Utilisation

Écrivons un programme complet montrant l'utilisation de la classe dérivée.

```
// herit_s.cpp

#include <iostream.h>
#include <string.h>

class personne
   {
   protected:
```

```
   char nom[30];
   char tel[20];
   public:
   void init(char *name, char *phone);
   void affiche(void);
   };

class eleve: public personne
   {
   protected:
   char promo[10];
   public:
   void init_p(char *classe);
   void affiche_p(void);
   };

void personne::init(char *name, char *phone)
   {
   strcpy(nom, name);
   strcpy(tel, phone);
   }

void personne::affiche(void)
   {
   cout << nom << " : " << tel;
   }
void eleve::init_p(char *classe)
   {
   strcpy(promo, classe);
   }
void eleve::affiche_p(void)
   {
   cout << promo;
   }

void main(void)
   {
   personne p;
   eleve e;
   p.init("Pierre", "01 23");
   e.init("Paul", "04 52");
   e.init_p("FI-2");
   cout << "\n";
   p.affiche();
   cout << "\n";
   e.affiche();
   cout << " : ";
   e.affiche_p();
   cout << "\n";
   }
```

5.2 Héritage multiple

Avec l'héritage simple, une classe peut hériter d'une autre classe et, éventuellement par transitivité, d'autres classes. Mais une classe ne peut pas hériter de deux classes (ou plus). Ceci peut se faire, par contre, grâce à l'héritage multiple.

5.2.1 Un exemple

Considérons, comme d'habitude, un petit exemple simple.

Le problème.- Supposons que nous voulions manipuler des points colorés, caractérisés par leurs coordonnées dans le plan et une couleur. Considérons deux classes de base `point` et `couleur` à partir desquelles nous dérivons une classe `pointcol`.

Les méthodes seront réduites à l'essentiel, à savoir une fonction d'initialisation et une fonction d'affichage pour chacune des classes. Pour simplifier, les couleurs seront des entiers compris entre 0 et 15.

Un programme.- Ceci conduit au programme suivant.

```
// herit_m.cpp

#include <iostream.h>

class point
   {
   protected:
   float x, y;
   public:
   void init(float, float);
   void affiche(void);
   };

class couleur
   {
   protected:
   short n;
   public:
   void init(short);
   void affiche(void);
   };

class pointcol: public point, public couleur
   {
   public:
   void init(float, float, short);
   void affiche(void);
   };

void point::init(float abs, float ord)
   {
```

```
   x = abs;
   y = ord;
   }

void point::affiche(void)
   {
   cout << '(' << x << ',' << y << ')';
   }

void couleur::init(short col)
   {
   n = col;
   }

void couleur::affiche(void)
   {
   cout << n;
   }

void pointcol::init(float abs, float ord, short col)
   {
   point::init(abs, ord);
   couleur::init(col);
   }

void pointcol::affiche(void)
   {
   point::affiche();
   cout << " de couleur ";
   couleur::affiche();
   }

void main(void)
   {
   pointcol p;
   p.init(2.0, 3.0, 4);
   cout << "\n";
   p.point::affiche();
   cout << "\n";
   p.affiche();
   cout << "\n";
   }
```

Remarques.- 1^o) Dans notre exemple très simple, on n'ajoute pas d'attributs supplémentaires. Ceci n'est évidemment pas toujours le cas.

- 2^o) Par risque de confusion, nous sommes obligés, lors de la définition de la méthode `pointcol::init()`, de préciser `point::init()` et `couleur::init()` en utilisant l'opérateur de portée. On n'est pas obligé de l'utiliser lorsqu'il n'y a pas ambiguïté.

5.2.2 Commentaires

Mise ne place.- Les classes de base sont définies comme d'habitude, sans oublier de remplacer certains « `private` » par « `protected` ». La classe dérivée indique les classes de base qu'elle grâce au suffixe de l'en-tête :

```
: public A, public B, public C
```

si elle hérite des classes `A`, `B` et `C`.

Résolution.- Dans notre exemple il y a trois méthodes `init()`, une par classe déclarée. On indique facilement à quelle classe chacune d'entre elles se rapporte grâce à l'opérateur de résolution de portée « :: ».

Cette façon de fairenous permet de définir la fonction `init()` de la classe `pointcol`.

L'objet `p` est déclaré comme appartenant à la classe `pointcol`. L'appel d'une méthode, par exemple `affiche()`, fait donc référence à `pointcol`::`affiche()` sans avoir à le spécifier, c'est-à-dire à utiliser explicitement l'opérateur de résolution de portée.

Cependant cet objet `p` appartient aussi à la classe `point`, puisque `pointcol` est une sous-classe de la classe `point`. On peut donc utiliser la fonction `affiche()` de cette classe pour `p` mais il faut alors spécifier explicitement `p.point`::`affiche()` pour lever toute ambiguïté. Ceci ne serait pas nécessaire pour une fonction dont le nom serait propre à une des classes de base.