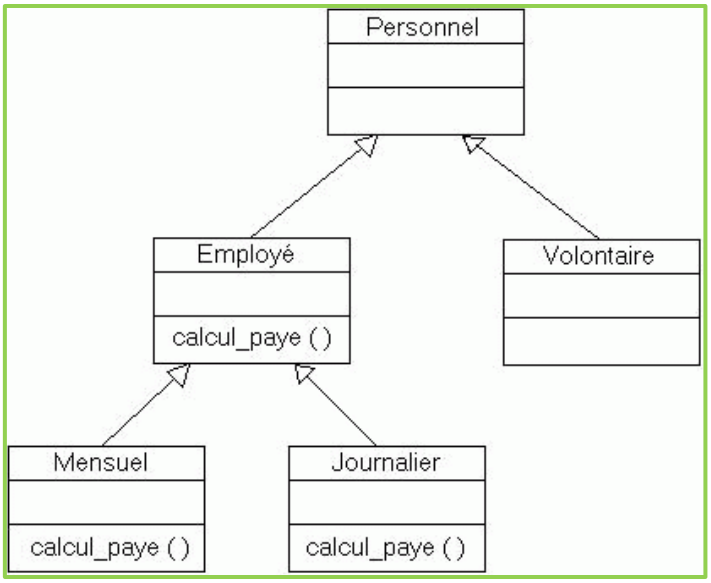
CM – Programmation C / C++

Chapitre 8 : Héritage et Polymorphisme



I- Principe d'héritage des classes :

On imagine que l'on souhaite écrire deux classes semblables mais qui possèdent quelques différences. Pour éviter de faire les mêmes déclarations dans les deux classes, on peut créer une classe qui regroupe tout ce qui est un commun, puis deux classes dérivées qui possèdent les différences. Lorsqu'on dérive une classe, la (ou les) nouvelle(s) classe(s) hérite(nt) de tous les attributs et de toutes les méthodes de la classe dont elle(s) est(sont) issue(s).

On peut aussi dériver une classe lorsque l'on souhaite ajouter des fonctionnalités à une classe existante sans avoir à réécrire le code qui la compose.

Exemple – Déclaration d'un classe « Vecteur » et de sa classe dérivée « Force » :

```
class Vecteur
{
    protected :

    double x;
    double y;
    double z;

    public :

    Vecteur();
    Vecteur(double a,double b,double c);

    double module();
};

class Force : public Vecteur
{
    double x0;
    double y0;
    double z0;

    public :

    Force();
    Force(double a,double b,double c,double a0,double b0,double c0);

    void affiche();
};
```

Ici, on déclare une classe nommée `Vecteur` composée de trois attributs protégés (accessibles uniquement par les classes dérivées) et de trois méthodes publiques (dont deux constructeurs). On déclare ensuite sa classe dérivée nommée `Force` qui possède trois attributs (eux aussi protégés) et trois méthodes publiques (dont deux constructeurs). La classe `Force` correspond ici à une force, qui contient un vecteur et un point d'appui. L'héritage est public, cela signifie que la zone publique de `Vecteur` est publique dans `Force`.

```
Vecteur::Vecteur()
{
    x=0;
    y=0;
    z=0;
}

Vecteur::Vecteur(double a,double b,double c)
{
    x=a;
    y=b;
    z=c;
}

double Vecteur::module()
{
    return sqrt(x*x+y*y+z*z);
}
```

```
Force::Force() : Vecteur()
{
    x0=0;
    y0=0;
    z0=0;
}

Force::Force(double a,double b,double c,double a0,double b0,double c0) : Vecteur(a,b,c)
{
    x0=a0;
    y0=b0;
    z0=c0;
}

void Force::affiche()
{
    cout << "Point d'appui : " << x0 << " " << y0 << " " << z0 << end;
    cout << "Force : " << x << " " << y << " " << z << endl;
    cout << "Module de la Force : " << module() << endl;
}
```

Les trois premières méthodes correspondent à celles contenues dans `Vecteur`. Les trois suivantes sont celles contenues dans `Force`. On écrit à côté de chacun des constructeurs de `Force` la commande `: Vecteur()`, qui permet à ses constructeurs d'appeler ceux de `Vecteur`. La méthode `affiche()` a accès aux attributs de , aux attributs de `Vecteur` et aux méthodes de `Vecteur` puisqu'elle est une méthode de la classe dérivée de `Vecteur`.

Dans cet exemple, on dit que `Vecteur` est la classe mère (ou parent) et `Force` est la classe fille (ou enfant). On dit aussi que `Force` hérite de `Vecteur`.

II- Méthodes virtuelles et polymorphisme :

Dans la mesure où une classe dérivée contient la classe de base, il est possible d'utiliser un pointeur sur la classe de base lorsqu'on travaille avec un objet dérivé.

```
class Parent
{
    public :

    void affiche()
    {
        cout << "Je suis dans la classe Parent." << endl;
    }
};

class Enfant : public Parent
{
    public :

    void affiche()
    {
        cout << "Je suis dans la classe Enfant." << endl;
    }

};
```

```
int main()
{
    Parent *a=new Parent;
    Enfant *b=new Enfant;
    Parent *c=new Enfant;

    a->affiche();
    b->affiche();
    c->affiche();

    return 0;
}
```

Ici, on déclare deux classes, une nommée `Parent` et une nommée `Enfant` qui hérite de la première. Elles possèdent chacune une méthode nommée `affiche()` qui ne renvoie rien et affiche un message pour déclarer la présence d'un pointeur dans sa classe. Dans le `main()`, on déclare trois pointeurs : deux pointant la classe `Parent` et un pointant la classe `Enfant`. On appelle ensuite, avec chaque pointeur, la méthode `affiche()` qui a le même nom pour chaque classe.

Lors de l'exécution, on s'attend à ce que les deux premiers pointeurs, en appelant la méthode, affichent le texte venant de leur type, donc de leur classe respective, ce qui est le cas. Cependant, le dernier pointeur, qui est de type `Parent` mais dont on crée l'objet dynamique de type `Enfant`, on s'attend à ce que le message d'appartenance à la classe `Enfant` s'affiche, mais c'est celui de `Parent` qui s'affiche à la place.

Pour que la méthode `affiche()` de la classe `Enfant` soit appelée lors de l'appel sur l'objet pointé par sa variable dynamique, on doit déclarer la méthode de la classe `Parent` dans le header en utilisant le mot clé **`virtual`**.

Le mécanisme de reconnaissance de la classe fille en utilisant des fonctions virtuelles est appelée polymorphisme. Il ne prend effet que lorsqu'on utilise des pointeurs sur une classe de base contenant une ou plusieurs fonctions virtuelles. Une fonction membre (ou méthode) devient virtuelle si on la déclare avec le mot clé **`virtual`** à l'intérieur de la définition de la classe, il n'est donc pas nécessaire de répéter ce mot clé dans la définition de la fonction en dehors de la classe.

Un constructeur ne peut pas être virtuel. Mais un destructeur virtuel est autorisé, parfois même indispensable selon les circonstances. Dans ce cas, pour deux classes parent et enfant qui possèdent un destructeur chacun, si le destructeur de la classe parent est virtuel, on appelle d'abord le destructeur de la classe enfant avant ce premier. Si les deux ne sont pas virtuels, seul le destructeur de la classe parent est appelé.

III- Méthodes virtuelles pures :

Il arrive que la présence d'une fonction virtuelle dans la classe de base soit nécessaire pour instituer le polymorphisme, mais qu'elle n'ait rien à faire de précis, comme si on pourrait s'en passer.

```
class Figure
{
    public :

    virtual double surface()
    {
        return 1.0;
    }
};

class Cercle : public Figure
{
    double Rayon;

    public :

    Cercle(double r)
    {
        Rayon=r;
    }

    double surface()
    {
        return 3.14159*Rayon*Rayon;
    }
};
```

Dans cet exemple, la méthode `surface()` de la classe `Figure` n'est pas fonctionnelle mais elle sert à instaurer le polymorphisme, c'est-à-dire de permettre au code `Figure *f=new Cercle; f->surface();` d'appeler la méthode `surface()` de la classe `Cercle`.

On peut éviter d'écrire une méthode dénuée de sens en la déclarant virtuelle pure. La syntaxe d'une méthode virtuelle pure est :

```
virtual type nom(arguments)=0;
```

Si on modifie la méthode `surface()` de la classe `Figure` de cette manière, elle deviendra virtuelle pure (on écrirait `virtual double surface()=0`). Le `=0` à la fin n'est pas une affectation mais permet de dire que la méthode est virtuelle pure. Les classes dérivées devront obligatoirement contenir la définition de cette méthode car aucun code n'y est associé.

Une classe qui possède une ou plusieurs fonctions virtuelles pures est qualifiée de classe abstraite. Leur utilisation implique des règles particulières :

- Les classes abstraites ne peuvent pas être instanciées, on ne peut pas créer de variable dynamique avec.
- Une classe abstraite ne peut pas servir comme type de paramètre ou comme valeur renvoyée par une fonction.
- Il est permis de référencer les classes abstraites par des pointeurs (ou des références si on les utilise comme paramètres). On peut déclarer une variable dynamique d'un type différent à la classe abstraite avec un pointeur de la classe abstraite.

Exemple – Programme avec une classe abstraite et deux classes dérivées :

```
class Figure
{
    public :

    virtual double surface()=0;
    virtual string nom()=0;
};
```

On déclare la classe abstraite `Figure` utilisée précédemment avec ses méthodes `surface()` et `nom()` qui sont virtuelles pures. Elles devront être définies dans les classes dérivées.

```
class Cercle : public Figure
{
    private :

    double Rayon;

    public :

    Cercle(double r);

    double surface();
    string nom();
};
```

```
Cercle::Cercle(double r)
{
    Rayon=r;
}

double Cercle::surface();
{
    return 3.14159*Rayon*Rayon;
}

string Cercle::nom()
{
    return "Cercle";
}
```

On déclare la classe dérivée `Cercle`, enfant de `Figure`. Elle est composée d'un attribut privé réel nommé `Rayon` et de trois méthodes dont un constructeur. Les deux autres méthodes, `surface()` et `nom()`, sont bien définies dans le fichier contenant les fonctions et donnent la surface de l'objet et son nom.

```
class Carre : public Figure
{
    private :

    double Cote;

    public :

    Carre(double c);

    double surface();
    string nom();
};
```

```
Carre::Carre(double c)
{
    Cote=c;
}

double Carre::surface()
{
    return Cote*Cote;
}

string Carre::nom()
{
    return "Carré";
}
```

On déclare la classe dérivée `Carre`, enfant de `Figure`. Elle est composée d'un attribut privé réel nommé `Cote` et de trois méthodes dont un constructeur. Les deux autres méthodes, `surface()` et `nom()`, sont bien définies dans le fichier contenant les fonctions et donnent la surface de l'objet et son nom.

```cpp
#include <iostream>
#include <vector>

using namespace std;

#include "Figures.h";


int main()
{

    Figure *Fig[2];
    Fig[1]=new Cercle(1.0);
    Fig[2]=new Carre(2.0);

    for(int i=0;i<2;i++)
        cout << "La surface du " << Fig[i]->nom() << "est" << Fig[i]->surface() << endl;

    int Reponse;
    cout << "Quel objet souhaitez-vous etudier ? : 1- Cercle  2- Carre" << endl;
    cin >> Reponse;
```

```cpp
    Figure *F;

    if(Reponse==1)
    {
        double r;
        cout << "Rayon =";
        cin >> r;

        F=new Cercle(r);
    }

    if(Reponse==2)
    {
        double c;
        cout << "Cote =";
        cin >> c;

        F=new Carre(c);
    }

    cout << "La surface du " << F->nom() << " est " << F->surface() << endl;

    return 0;
}
```

Dans le `main()`, on déclare un tableau dynamique de `Figure` de deux cases : la première est une variable dynamique de type `Cercle` initialisée à `1.0` et la seconde est une variable dynamique de type `Carre` initialisée à `1.0` (le type de l'objet sera défini à l'exécution pour les cases du tableau). On crée ensuite une boucle **`for`** pour afficher les objets du tableau, leurs noms et leurs surfaces en appelant les méthodes `surface()` et `nom()`.

On déclare après une variable entière nommée `Reponse` dont on demande à l'utilisateur de choisir une valeur entre 1 et 2 pour choisir un objet à étudier. On déclare un nouveau pointeur de `Figure` nommé `F` et on crée deux **`if`** correspondant à chacun des cas étudiés. Dans les deux cas, on déclare une variable réelle dont l'utilisateur entre la valeur et dont on crée un objet dynamique grâce à `F` qui sera du type de l'objet choisi. Si la valeur entrée est 1, on étudie un cercle ; si la valeur entrée est 2, on étudie un carré.

On affiche ensuite les caractéristiques de l'objet choisi, en affichant son nom et sa surface en appelant les méthodes `surface()` et `nom()`.