Chapitre 13
Le polymorphisme en C++

1- Redéfinition des fonctions

✓**Le C++ offre la possibilité de redéfinir une fonction membre de la classe de base dans la classe dérivée, comme la méthode** affiche() **dans l'exemple suivant.**

```
class Personne
{    public:
        void affiche();
       ..................;
} ;
class Employe: public Personne
{    public:
        void affiche();
       ..................;
} ;
void Personne::affiche()
{
   cout<<"Personne"<<endl;
}
void Employe::affiche()
{
   cout<<"Employe"<<endl;
}
```

✓ **Considérons maintenant le programme suivant :**

```
int main()
{ Personne *P=new Employé;
  P->affiche();
  delete P;
  Employe *E=new Employé;
  E->affiche();
  delete E;
}
```

✓ **Ce programme affiche :**

Personne
Employe

✓**Attention, un pointeur de base peut contenir les adresses des objets dérivés mais jamais l'inverse !**

✓**Ici,** P **est un pointeur de base sur** Personne **qui contient l'adresse d'un objet de type** Employe**.**

✓**Ce petit programme met en évidence que la méthode** affiche() **appelée est toujours celle correspondant au type du pointeur.**

✓**Lorsqu'un objet de type** Employe **est désigné par un pointeur de type** Employe, **la méthode** affiche() **appelée est celle de la classe** Employe**; tandis que si le même objet de type** Employe **est désigné par un pointeur de type** Personne, **la méthode** affiche() **appelée est celle de la classe** Personne.

2- Fonctions polymorphes ou virtuelles

✓**On aurait espéré dans l'exemple précédent que la méthode** affiche() **appelée soit, non celle désignée par le type du pointeur, mais celle désigné par le type de l'objet ; c'est-à-dire qu'un objet de type** Employe **affiche « Employe » même si on utilise pour le désigner un pointeur de base !**

✓**Cela est possible à condition d'utiliser des fonctions dites** polymorphes **ou** virtuelles**.**

✓**Une fonction virtuelle est une fonction qui est appelée en fonction du type d'objet et non pas en fonction du type de pointeur utilisé.**

✓**Pour déclarer qu'une fonction membre est polymorphe, il suffit de faire précéder son prototype par le mot clé *virtual*.**

✓**Il n'est pas nécessaire de faire apparaître ce mot clé devant le corps de la méthode.**

✓**Nous reprenons l'exemple précédent en y adjoignant la classe** Fonctionnaire.

```
class Personne
{     public:
          virtual void affiche();
         ..................;
} ;
class Employe: public Personne
{     public:
         virtual  void affiche();
         ..................;
} ;
class Fonctionnaire: public Personne
{     public:
         virtual  void affiche();
         ..................;
} ;
void Personne::affiche()
{
    cout<<"Personne"<<endl;
}
void Employe::affiche()
{
    cout<<"Employe"<<endl;
}
void Fonctionnaire::affiche()
{
    cout<<"Fonctionnaire"<<endl;
}
```

✓ **Le programme suivant utilise un pointeur de base sur** Personne **pour désigner des objets de type** Employe **ou** Fonctionnaire

```
int main()
{
Personne* E= new Employe;
E->affiche();
delete E;
Personne* F= new Fonctionnaire;
F->affiche();
delete F;
}
```

✓ **Et affiche :**

Employe
Fonctionnaire

✓**Techniquement, seule la méthode *affiche()* de la classe *Personne* doit être déclarée virtuelle.**

✓**Cependant, il est conseillé d'indiquer le mot clé *virtual* pour toutes les méthodes concernées par le polymorphisme, et cela quelque soit leur niveau dans la hiérarchie des classes.**

✓**Lorsque l'on appelle une fonction virtuelle pour un objet, le C++ cherche cette méthode dans la classe correspondante.**

✓**Si cette méthode n'existe pas dans la classe concernée, le C++ remonte la hiérarchie des classes jusqu'à ce qu'il trouve la fonction appelée.**

✓**La résolutions des appels des fonctions virtuelles à lieu à l'exécution (*ligature dynamique*) et non pas à la compilation comme pour les autres fonctions.**

3- Appel d'une fonction polymorphe dans sa propre classe

✓**On peut souhaiter utiliser une fonction polymorphe dans sa propre classe, ce qui permet une factorisation d'un code très efficace.**

✓**Par exemple, si l'on veut consulter tous les personnes, il suffit d'afficher un message ainsi que le type de la personne concernée, il va être intéressant de définir une méthode *Consulter* () dans la classe *Personne* utilisant la fonction polymorphe *Afficher()*.**

✓**Comme on rappelle qu'une fonction polymorphe est appelée en fonction du type de l'objet, les employés et les fonctionnaires sont consultés selon leurs types respectifs, ce qui est tout à fait remarquable !**

```
class Personne
{    public:
virtual void affiche();
        void Consulter()
};
class Employe: public Personne
{    public:
virtual void affiche();
};
class Fonctionnaire: public Personne
{    public:
virtual void affiche();
};
void Personne::affiche()
{
   cout<<"Personne"<<endl;
}
void Personne::Consulter()
{
   cout<<"Je suis :"<<endl;
   affiche();
}
void Employe::affiche()
{
   cout<<"Employe"<<endl;
}
void Fonctionnaire::affiche()
{
   cout<<"Fonctionnaire"<<endl;
}
```

Le programme suivant :

```
int main()
{
 Personne* E= new Employe;
 E->consulter();
 delete E;
 Personne* F= new Fonctionnaire;
 F->consulter();
 delete F;
}
```

Affiche correctement :

Je suis
Employe
Je suis
Fonctionnaire

✓**En vérité, tout cela n'est possible que grâce à l'utilisation, que fait C++ de la variable *this*, qui permet de désigner l'objet courant par lequel la méthode est appelée.**

✓**D'ailleurs, le compilateur modifie le code en passant *this* comme un argument caché :**

```
void Personne::Consulter(Employe*this)
{
   cout<<"Je suis :"<<endl;
   this->affiche()
}
```

Remarques :

- Un constructeur ne peut pas être virtuel,

- Un destructeur peut-être virtuel,

- La déclaration d'une fonction membre virtuelle dans la classe mère, sera comprise par toutes les classes descendantes (sur toutes les générations).

4- Appel de la fonction membre d'une classe de base dans le cadre du polymorphisme

✓**Parfois, il peut être intéressant d'appeler une fonction de la classe de base pour la compléter plutôt que pour la substituer complètement.**

✓**Dans ce cas, il convient d'appeler la fonction virtuelle d'origine au sein de la nouvelle méthode.**

✓**Chaque fois que l'on souhaite utiliser une fonction membre de la classe de base, il convient d'utiliser le nom complet de cette dernière (**NomClasse::NomFonction**) si cette fonction existe également dans la classe dérivée.**

✓**Considérons l'exemple suivant avec la méthode** affiche() **de la classe** Personne **qui est complétée dans ses classes dérivées.**

```
class Personne
{    public:
virtual void affiche();
 } ;
class Employe: public Personne
{    public:
virtual void affiche();
} ;
class Fonctionnaire: public Personne
{    public:
virtual void affiche();
 } ;
void Personne::affiche()
{
   cout<<" Je suis Personne"<<endl;
}
void Employe::affiche()
{
   Personne::affiche();
   cout<<"Employe"<<endl;
}
void Fonctionnaire::affiche()
{
   Personne::affiche();
   cout<<"Fonctionnaire"<<endl;
}
```

Le programme suivant :

```
int main()
{
Personne* E= new Employe;
E->affiche();
delete E;
}
```

Affiche correctement :

Je suis Personne
Employe

5- Polymorphisme et destructeur

✓**Si l'on définit un destructeur dans le cadre du polymorphisme, il faut toujours le déclarer en tant que fonction virtuelle.**

```
virtual ~Personne() ;
```

✓**Plus précisément, dans le cas d'une hiérarchie de classes, cela oblige à définir un destructeur virtuel dans les classes de base !**

✓**Si jamais cette contrainte n'était pas respectée, le programme risque de ne pas libérer toute la mémoire utilisée.**

6- Fonctions virtuelles pures et classes abstraites

La *fonction virtuelle pure* est une fonction virtuelle pour laquelle le prototype est suivi de l'expression *=0*.

L'instruction suivante montre la définition d'une fonction virtuelle pure :

```
virtual void affiche()=0 ;
```

Définir une fonction virtuelle pure (équivalent à des méthodes abstraites en Java**) dans une classe a deux incidences :**

- **Une classe qui contient la déclaration d'une fonction virtuelle pure devient *une classe abstraite*.**
- **La redéfinition des fonctions virtuelles pures est obligatoire dans toutes les classes dérivées. Dans le cas contraire, les classes deviennent également abstraites.**

✓**L'avantage de ces fonctions membres consiste donc à obliger les classes dérivées à les redéfinir sous peine de devenir elles-mêmes abstraites.**

Reprenons l'exemple de la classe Personne.

```
class Personne
{    public:
        virtual  void affiche()=0;
 } ;
class Employe: public Personne
{    public:
        virtual  void affiche();
} ;
class Fonctionnaire: public Personne
{    public:
        virtual void affiche();
 } ;
void Employe::affiche()
{

   cout<<"Je suis Employe"<<endl;
}
void Fonctionnaire::affiche()
{

   cout<<"Je suis Fonctionnaire"<<endl;
}
```

✓ **Les fonctions virtuelles pures sont les seules méthodes qui peuvent ne pas avoir de corps de fonction.**

✓**Cela n'est possible qu'à la condition expresse que la méthode ne soit pas appelée directement !**

✓**Une classe devient abstraite si elle possède au moins une fonction virtuelle pure.**

✓**Dans notre exemple la classe** Personne **est abstraite ; et de ce fait il devient** impossible de créer un objet directement à partir de cette classe !