Chapitre X : Polymorphisme

X.1 Etude de Cas [2]

Etudions l'exemple suivant :

```
class A
{   private :
       …..
    public :
       ….
       void f()
       ….
} ;
void A :: f()
{ cout<< "A::f()" <<endl;  }
```

```
class B: public A
{     …
  public:
     void f();
     …
};
void B :: f()
{  cout<< "B::f()" <<endl; }
```

```
void main()
{    A a;
     B b;
     A *p;
     P= &a;  p->f();    // affiche A::f()
     p=&b;   p->f();    // affiche A::f() aussi
}
```

Remarques:

✍ Le choix de la fonction est conditionné par le type de **p** connu au moment de la **compilation**

Puisque p est un A* alors il utilise la méthode A ::f()

Il sera mieux que le deuxième appel écrit B ::f() ? Puisque p contient un B

⇒ Solution : ***Fonction virtuelle***

X.2. Fonction virtuelle [2]

```
class A
{   private :
       …..
    public :
       ….
       virtual void f()
       ….
} ;
```

```
void A ::f()
{ cout<< "A::f()" <<endl; }
```

```
class B: public A
{     …
   public:
      void f();
       …
};
void B::f()
{ cout<< “B::f()” <<endl; }
```

```
void main()
{    A a;
     B b;
     A *p;
     P= &a;  p->f(); // affiche A::f()
     p=&b;   p->f(); // affiche B::f() aussi
}
```

Remarques :

Le choix de la fonction est **maintenant** conditionné par le type exact de l'objet pointé par p connu au moment de **l'exécution** puisque p nous emmène sur un B* alors on utilise B ::f()

X.3 Définition de Polymorphisme

Mécanisme qui consiste à définir des fonctions de même nom dans les classes de bases et les classes dérivées et qui répondent différemment à un même appel. Les classes dérivées héritent tous les membres publics et protégés de la classe mère.

- **En pratique** : Déclarer **virtual** la fonction concernée dans la classe la plus générale de la hiérarchie d'héritage. (Dans l'exemple précédent A)
- Toutes les classes dérivées qui apportent une nouvelle version de f() utiliseront leur fonction à elles.
- Il reste évidement possible d'appeler la fonction f() de A en spécifiant p->A ::f(), mais à ce moment là pourquoi avoir utilisé une fonction virtuelle !?

X.4 Destructeur virtuel [2]

Considérons l'exemple suivant :

```
class A
{   protected :
      int *p;

   public :
     A()
      { p=new int[2] ;
        cout<<"A()"<<endl ;
      }
```

```
   ~A()
   { delete [] p ;
     cout<<"~A()"<<endl ;
    }
} ;

class B : public A
{    int *q;
   public :
    B()
    { p=new int[20] ;
      cout<<"B()"<<endl ;
    }
    ~B()
   { delete [] p ;
     cout<<"~B()"<<endl ;
   }
} ;

void main()
{
 for (int I =0; I<4;I++)
   {  A *pa = new B();
      delete pa;
   }
}
```

Affiche ceci :

```
A()
B()
~A()
A()
B()
~A()
A()
B()
~A()
A()
B()
~A()

Process returned 0 (0x0)   execution time : 3.212 s
Press any key to continue.
```

✍ On doit faire appel au destructeur de B avant celui de A, puisque **pa** référence un ***B.**
Dans ce cas on a un espace mémoire alloué non libéré : (fuite de mémoire)

⇒ Solution : **destructeur virtuel**

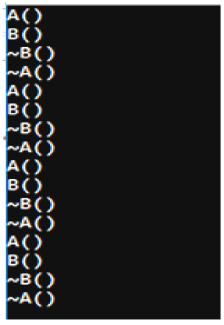
La classe A devient ainsi :

```
class A
{  protected :
     int *p;

  public :
    A()
    { p=new int[2] ;
      cout<<"A()"<<endl ;
    }
```

```
    virtual ~A()
    { delete [] p ;
      cout<<"~A()"<<endl ;
     }
} ;
```

A l'exécution, on a ceci:



Le bon destructeur est appelé.

X.5 Fonction virtuelle pure – classe abstraite [2]

En POO, nous pouvons définir des classes destinées non pas à instancier des objets, mais simplement à donner naissance à d'autres classes par héritage. On dit qu'on a affaire à des ***«Classes Abstraites».***

En C++, nous pouvons toujours définir de telles classes, en déclarant des fonctions membres virtuelles dont on ne précise pas le contenu dans la classe de base. Seules les classes de base possèdent alors, éventuellement une description du corps de ces fonctions virtuelles. On les appelle des fonctions virtuelles pures.

⇒ Une ***fonction virtuelle pure*** se déclare en remplaçant le corps de la fonction par les symboles = 0.

Syntaxe :

```
class NomClasse
{
   // …
   virtual TypeRetout nomFonction (liste des arguments)=0;
  //…
}
```

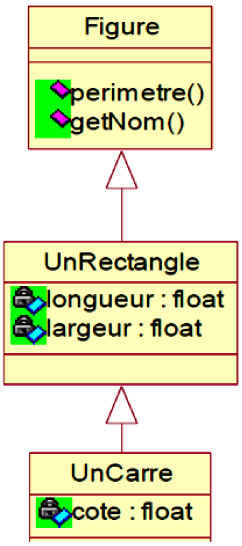
Remarque:

- Une classe comportant au moins une fonction virtuelle pure est considérée comme abstraite et il n'est plus possible de déclarer des objets de son type.

- Une fonction déclarée virtuelle pure dans une classe de base ***doit obligatoirement être redéfinie*** dans une classe dérivée ou déclarée à nouveau virtuelle pure ; dans ce dernier cas, la classe dérivée est aussi abstraite.

Exemple:

```
class Figure
{
  Public:
    virtual float perimetre()=0;
    virtual float surface()=0;
    virtual void dessiner();
}
```



✍ La classe figure est une abstraction. Un programme ne créera pas d'objet *Figure*, mais des objets *Rectangle*, *Cercle*, etc..
On remarque que la fonction *perimetre()* introduite au niveau de la classe *Figure* n'est pas un service rendu aux programmeurs mais une contrainte :

- son rôle n'est pas de dire ce qu'est le périmètre d'une figure, mais ***d'obliger les futures classes dérivées de figure à le dire.***

```
class Figure
{  public:
     virtual float perimetre()=0;
     virtual char * getNom()=0;
};

class UnRectangle:public Figure
{     float longueur;
      float largeur;

  public :
      UnRectangle(float longueur, float largeur)
      {
          this->longueur=longueur;
          this->largeur=largeur;
      }

     float perimetre()
     {  return 2*(longueur+largeur); }

     char* getNom()
     {  return "RECTANGLE"; }
};

class UnCarre:public UnRectangle
{      float cote;

   public:
      UnCarre(float cote):UnRectangle(cote,cote)
      {  this->cote=cote;  }

    float perimetre()
    {  return cote*4;  }

   char* getNom()
   {   return "CARRE";  }
};
```

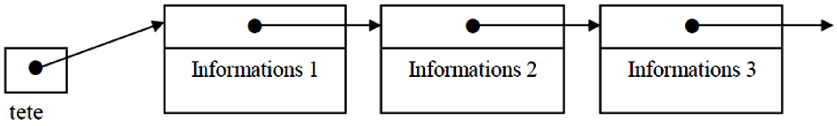
```
void main()
{
   UnRectangle *Rect=new UnRectangle(10,5);
   UnCarre *Carre=new UnCarre(10);
  Figure *T[2];
  T[0]=Rect;
  T[1]=Carre;
  for (int i=0;i<=1;i++)
  {
    cout<<"Le PERIMETRE DU "<<T[i]->getNom()<<" = "<< T[i]->perimetre()<<endl;
  }
}
```

X.6 Exemple d'utilisation de fonctions virtuelles : liste hétérogène [7]

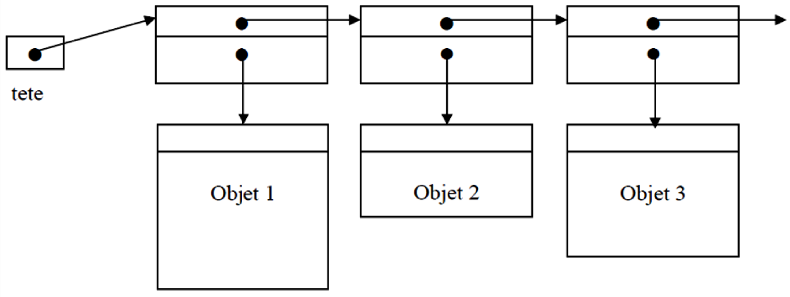
Nous allons créer une classe permettant de gérer une liste chaînée d'objets de types différents et disposant des fonctionnalités suivantes :

- ajout d'un nouvel élément,
- affichage des valeurs de tous les éléments de la liste,
- mécanisme de parcours de la liste.

Rappelons que, dans une liste chaînée, chaque élément comporte un pointeur sur l'élément suivant.



Mais ici l'on souhaite que les différentes informations puissent être de types différents. Aussi cherchons-nous à isoler dans une classe (nommé *liste*) toutes les fonctionnalités de gestion de la liste elle-même sans entrer dans les détails spécifiques aux objets concernés. Nous appliquerons alors ce schéma :



La classe *liste* est composée de :

- un pointeur sur l'élément suivant
- un pointeur sur l'information associée (en fait, ici, un objet).

```
struct element
{   element *suivant ;
    mere *contenu ;
} ;
```

```
class Liste
{    element *tete ;
  public:
     Liste() ;
    ~Liste() ;
    void ajouter(mere *) ;
    void afficher() ;
     .....
};
```

Exemple :

```
class Article
{    long code;
    char nom[100];
    double quantite;
    double prix;

 public:
    virtual void afficher()
    {   cout <<"code="<<code<<endl ;
        cout <<"Nom="<<nom<<endl ;
        cout <<"Quantité="<<quantite<<endl ;
        cout <<"Prix unitaire="<<prix<<endl ;
     }

    virtual void saisir()
    {  cout <<"code="<<endl;              cin>>code ;
       cout <<"Nom="<< endl ;             cin >> nom ;
       cout <<"Quantité="<< endl ;       cin >> quantite ;
       cout <<"Prix unitaire="<< endl ;  cin>> prix ;
     }
};

class Boissons:public Article
{   public:
      double volume;

     void afficher()
      {   cout << "c'est un boissons"<<endl ;
          Article ::afficher() ;
          cout << "volume="<<volume<<endl ;
       }

     void saisir()
     {   cout << "saisie d'un boissons"<<endl ;
         Article ::saisir() ;
         cout << "Volume= "<<endl ;   cin>> volume ;
     }
} ;

class Confiture: public Article
{  public:
    double poids;
```

```
      void afficher()
      {  cout << "c'est un confiture"<<endl ;
         Article ::afficher() ;
         cout << "Poids="<<poids<<endl ;
      }

      void saisir()
      {  cout << "c'est un confiture"<<endl ;
         Article ::saisir() ;
         cout << "Poids= "<<endl ;   cin>> poids ;
      }
} ;

typedef struct element
{   element *suivant ;
    Article *contenu ;
} ;

class Liste
{    element *tete ;
  public :
      Liste()
      { tete=NULL ;}

     ~Liste()
     { delete tete ;}

     void ajouter(Article *a) // au début de la liste
     {   element *nouv= new element ;
         nouv->contenu=a ;
         nouv->suivant=tete ;
         tete=nouv ;
     }

     void afficher()
     {  elemnet *parc=tete ;
        while(parc !=NULL)
        {  parc->contenu->afficher() ;
           parc=parc->suivant ;
        }
     }
} ;

void main()
{
    Confiture c ;
    c.saisir() ;
    Boissons b ;
     b.saisir() ;
     Liste L ;
     L.ajouter(&c ) ;
     L.ajouter(&b ) ;
     L.afficher() ; //affichage de toute la liste
}
```