Chapitre 6 : La technique de l'héritage

1. Mise en œuvre de l'héritage

1.1. Exemple simple sans constructeur ni destructeur

1.1.1. La classe de base 'point' définie dans le fichier d'entête 'point.h'

```
class point
{
        int x, y;
     public:
        void initialise(int,int);
        void deplace(int,int);
        void affiche();
};
void point::initialise(int a,int b){x=a; y=b;}
void point::deplace(int  a,int b){x=x+a; y=y+b;}
void point::affiche(){cout<<"Point : "<<x<<" - "<<y<<endl;}
```

1.1.2. La classe 'point_colore' derivée de la class 'point'

```
#include <point.h>
Class point_colore : public point
{
        int couleur;
     public:
        void colorer(int c) {couleur=c;}
};
```

1.2. Commentaire

- La déclaration 'class point_colore : public point' spécifie que la classe 'point_colore' est dérivée de la classe de base 'point'.

- Le mot clé 'public' signifie que les membres publics de la class de base seront des membres publics de la classe dérivée.

Le programme principal

```
main()
{
     point_colore   p;
     p.inisialise(12,27);     p.colorer(7);       p.affiche();
     p.deplace(1,2);          p.affiche();
}
```

Exécution

```
Point : 12 - 27
Point : 13 - 29
```

2. Utilisation, dans une classe dérivée, des membres de la classe de base

Après avoir fait appel à la fonction 'colorer', la fonction 'affiche' ne donne aucune information sur la couleur d'un point. Ce qui nous conduit donc à créer une fonction 'affiche_c' membre de 'point_colore pour afficher les coordonnées x, y et la couleur c.

```
void affiche_c()
{
     cout<<"Point : "<<x<<" - "<<y;
     cout<<"En couleur : "<<couleur<<endl;
}
```

Or, une classe dérivée n'a pas accès aux membres privés de la classe de base. La solution est donc :

```
void affiche_c()
{ affiche(); cout<<" en couleur : "<<couleur<<endl; }
```

De même on peut initialiser x, y et couleur en même temps avec une fonction 'initialise_c' de la classe dérivée 'point_colore'.

Le programme complet est donc :

```
class point
{
       int x, y;
     public:
       void initialise(int a,int b){x=a;y=b;}
       void deplace(int a,int b){x=x+a; y=y+b;}
       void affiche(){cout<<"Point : "<<x<<" - "<<y;}
} ;

//-----------------------------------------------------
class point_colore:public point
{
       int couleur;

     public:
       void initialise_c(int c)
       {couleur=c;}
       void colorer(int c)
       {couleur=c;}

       void affiche_c()
       {
         affiche();
         cout<<" En couleur : "<<couleur<<endl;
       }
};
```

```
main()
{
     point_colore   p;
     p.initialise(12,9); p.colorer(7);
     p.affiche(); cout<<endl; p.affiche_c();
     p.deplace(1,2);p.affiche(); cout<<endl;  p.affiche_c();
}
```

Exécution

3. Redéfinition des fonctions membres

Les méthodes 'affiche' de la classe 'point' et 'affiche_c' de la classe 'point_colore' font un travail analogue. De même pour les fonctions 'initialise' et 'initialise_c'.
En c++, il est possible de redéfinir la fonction 'affiche' ou la fonction 'inialise' pour la classe dérivée.

Exemple

```
#include <point.h>
class point_colore:public     point
{
       int couleur;

     public:
       void colorer(int c){couleur=c;}
       void affiche()
       {
          point::affiche();
          cout<<" en couleur : "<<couleur<<endl;
       }
       void initialise(int a,int b,int c)
       {point::initialise(a,b); couleur=c;}
};

//-----------------------------------------
main()
{
     point_colore p;
     p.initialise(12,27,9);
     p.affiche();
     p.deplace(10,20);   p.affiche();
     p.colorer(7);       p.affiche();
}
```

Exécution

4. Appel des constructeurs et des destructeurs

Si on a :

```
class     point
{
       int x,y;
     public:
       point(int,int);
};
//-----------------------------------------------
class point_colore:public point
{
       int couleur;
     public:
       point_colore(int,int,int);
};
```

Pour créer un objet de la classe 'point_colore', il faut tout d'abord créer un objet de la classe 'point', donc faire appel au constructeur de la classe 'point', le compléter par ce qui est spécifique a la classe 'point_colore' et faire appel au constructeur de la classe 'point_colore'.

Si on souhaite que le constructeur de la classe 'point_clolre' retransmette au constructeur de la classe 'point' les premières informations reçues, on écrira :
'point_colore(int a, int b, int c):point(a, b) ;'.
Ainsi, la déclaration : 'point_colore(2,4,5) ;' entraînera :

- l'appel du constructeur 'point' qui recevra les valeurs 2 et 4.
- l'appel du constructeur 'point_colore' qui recevra les valeurs 2, 4 et 5.

Il est toujours possible de mentionner les valeurs par défaut :
point_colore(int a=0, int b=2, int c=5):point(a,b)
Donc la déclaration 'point_colore(17) ;' entraîne :
- appel du constructeur 'point' avec les valeurs 17 et 0.
- appel du constructeur 'point_colore' avec les valeurs 17, 2 et 5.

Exercice

Reprendre le programme précédent en ajoutant les constructeurs et les destructeurs correspondants, tout en affichant les moments de constriction et de destruction des objets. Prévoir aussi des objets dynamiques.

D'une manière générale

Si la classe de base ne possède pas de constructeur, aucun problème particulier ne se pose. De même si elle ne possède pas de destructeur.
En revanche, si la classe dérivée ne possède pas de constructeur, alors que la classe de base en comporte, le problème sera posé lors de la transmission des informations attendues par le constructeur de la classe de base. La seule situation acceptable est celle où la classe de base dispose d'un constructeur sans arguments.
Lors de la transmission d'information au constructeur de la classe de base, on peut utiliser des expressions ou des arguments.

Exemple

```
pointcolore(int a=5, int b=5, int c=4):point(a*2, b*5);
```

5. Contrôle des accès

5.1. L'héritage privé

Pour que l'utilisateur d'une classe dérivée n'ait pas accès aux membres publics de sa classe de base, il suffit de remplacer le mot 'public' par 'private' dans sa déclaration.

Exemple

```
class     point
{
       int     x, y;
     public:
       void initialise(int,int) {x=a;   y=b;}
       void deplace(int,int) {x=x+a; y=y+b;}
       void affiche(){   ……….   }
};
//---------------------------------------
class point_colore : private point
{
       int     couleur;
     public:
       void    colorer(int c)
       {couleur=c;}
};
```

Si on a :

```
point_colore O(12,4,6);
```

Les appels suivants sont rejetés :

```
O.affiche(); ou O.point::affiche();
O.deplace(1,5); ou O.point::depace(1,5);
```

Remarque

Cette technique de fermeture d'accès à la classe de base ne sera employée que dans des cas précis, lorsque par exemple toutes les fonctions utiles de la classe de base sont redéfinies dans la classe dérivée, et qu'il n'y a aucune raison de laisser l'utilisateur accéder aux anciennes.

5.2. Les membres protégés d'une classe

Les membres protégés restent inaccessibles à l'utilisateur (comme les membres privés). Mais ils seront accessibles aux membres d'une éventuelle classe dérivée, tout en restant inaccessibles aux utilisateurs de cette classe.

Exemple

Supposons qu'on a :

```
class     point
{
     protected:
       int x,y;
     public:
       void initialise(int,int);
       void deplace(int,int);
       void affiche();
};
```

On peut donc déclarer dans la classe 'point_colore' dérivée de la classe 'point' une fonction 'affiche' qui accède aux membres protégés x et y.

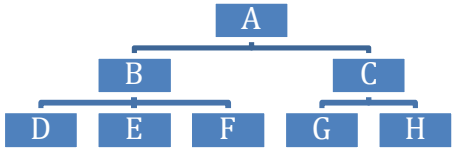
```
class     point_colore:public point
{
       int     couleur;
     public:
       void affiche()
       { cout<<"Point coloré " <<x<<" - " <<y<<endl;
         cout<<"en couleur : "<<couleur<<endl;
       }
};
```

Remarque

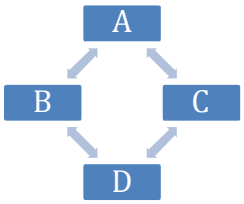
Lorsqu'une classe dérivée possède une classe amie, cette dernière dispose des mêmes droits d'accès que les fonctions membres de la class dérivée.

6. L'héritage en général

D'une façon générale, l'héritage peut être représenté par les arbres suivants :



Héritage simple



Héritage complexe

7. Conversion d'un objet dérivé dans un objet d'un type de base

Si on a :

```
point          o1;
point_colore   o2;
```

- l'affectation ‘o1=o2 ;’ est juste.
- l'affectation ‘o2=o1 ;’ est rejetée (si o2 a des arguments définis par défaut, on n’aura pas de problème).

Exercice

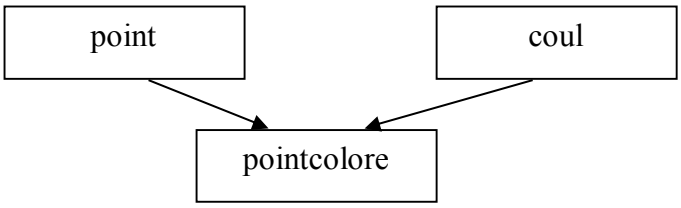
A. Ecrire un programme écran de veille affichant des points (objets de type 'pointg' à définir) en couleur à tout endroit de l'écran, et ce, jusqu'a ce que l'utilisateur appuie sur une touche quelconque.

B. Reprendre le même programme en ajoutant des objets de la classe 'cercle' dérivée de la classe 'pointg' et ayant en plus, la donnée membre 'rayon' et les méthodes :
- affiche_cercle : pour afficher un cercle aux coordonnées x, y.
- cache_cercle : pour rendre un cercle invisible.
- zoom_cercle : pour afficher une série de cercles concentriques.
- deplace_cercle : pour déplacer un cercle.
- ……..

Chapitre 7 : L'héritage multiple

1. Mise en œuvre de l'héritage multiple

L'exemple suivant met en évidence la notion d'héritage multiple à travers la classe 'point_colore' héritant des classes 'point' et 'coul'.




```
class     point
{
     int  x,y;
     public:
          point(int a,int b)
          {
               x=a ;y=b ;
               cout<<"Constructeur de point ";
          }
          ~point(){cout<<"Destructeur de point ";}
          void affiche(){cout<<"point "<<x<<" - "<<y;}
};

//----------------------------
class coul
{
       int  couleur ;
     public :
       coul(int c)
       {
          couleur=c ;
          cout<<"Construction de coul ";
       }
       ~coul(){cout<<"Destruction de coul ";}
       void    affiche()
       {cout<<"Couleur : "<<couleur<<endl;}
};

//-------------------------------------------------
class point_colore:public point,public coul
{
    public:
      point_colore(int a,int b,int c):point(a,b),coul(c)
      {cout<<"construction de pointcolore\n";}
```

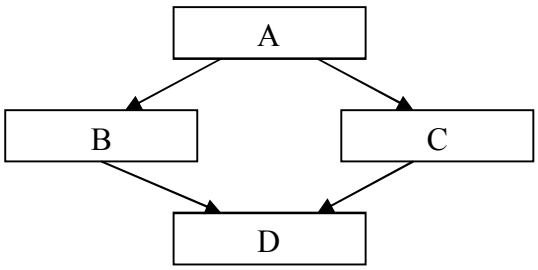
```
        ~pointcolore()
        {cout<<"destruction de pointcolore\n"; }
        void affiche()
        {   point::affiche();
            coul::affiche();
        }
};
//---------------------------------------
main()
{
     point_colore   o(100,200,3);
     o.affiche();
     o.point::affiche();
     o.coul::affiche();
}
```

Exécution

```
Constructeur de point Construction de coul construction de pointcolore
point 100 - 200Couleur : 3
point 100 - 200Couleur : 3
```

2. Les classes virtuelles

Si on a :



Donc les déclarations suivantes :

```
class     A
{
     int x,y;
     ……….
};
class     B:public A
{  ……….  };
class     C:public A
{  ……….  };
class     D:public B, public C
{  ……….  };
```

Impliquent que la classe 'D' hérite deux fois de la classe 'A', donc les membres de 'A' vont apparaître deux fois dans 'D'.
- les fonctions membres ne sont pas réellement dupliquées.
- les données membres seront effectivement dupliquées.

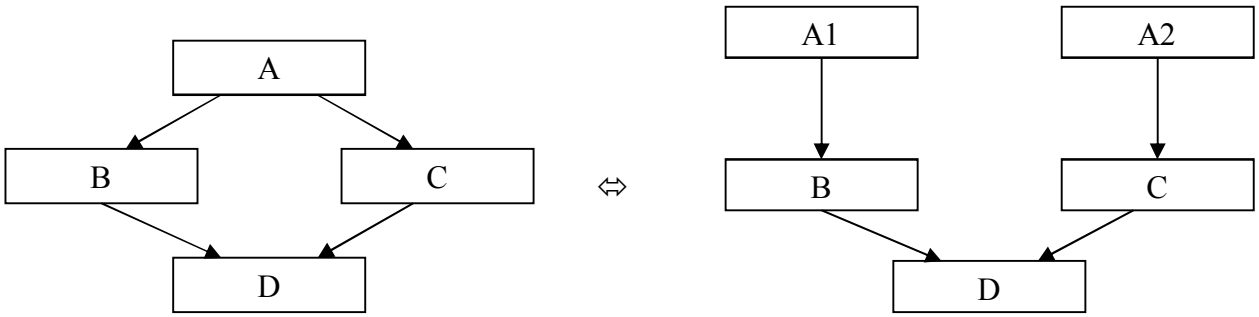
Donc :
- si on veut laisser cette duplication, on utilise :
  A::B::x ≠ A::C::x ou B::x ≠ C ::x
- si on ne veut pas de cette duplication, on précisera la classe 'A' comme classe virtuelle dans les déclarations des classes 'B' et 'C'.

```
class     A
{
     int  x,y;
     ……..
};
class     B:public virtual A
{
     ……..
};
class     C:public virtual A
{
     ……..
};
class     D:public B, public C
{
     ……..
};
```

Remarque

- le mot-clé 'virtual' apparaît dans la classe 'B' et la classe 'C'.
- le mot-clé 'virtual' peut être placé avant ou après le mot-clé 'public'.

3. Appel des constructeurs et des destructeurs dans le cas des classes virtuelles



- Si 'A' n'est pas déclarée 'virtual' dans les classes 'B' et 'C', les constructeurs seront appelés dans l'ordre : 'A1', 'B', 'A2', 'C' et 'D'.
- Si 'A' a été déclarée 'virtual' dans 'B' et 'C', on ne construira qu'un seul objet de type de 'A' (et non pas deux objets).

Quels arguments faut-il transmettre alors au constructeur ? Ceux prévus par 'B' ou ceux prévus par 'C' ?
Le choix des informations à transmettre au constructeur de 'A' se fait, non plus dans 'B' ou 'C', mais dans 'D'. Pour ce faire, C++ autorise (uniquement dans ce cas) à spécifier, dans le constructeur de 'D', des informations destinées à 'A'.

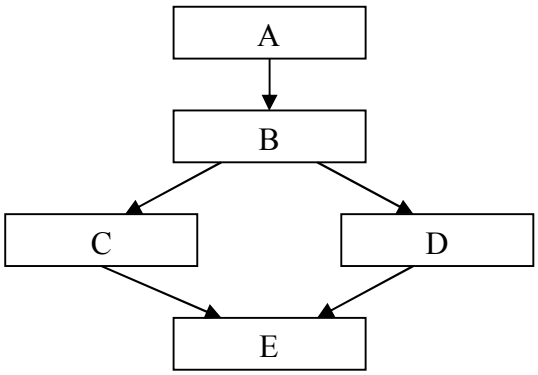
Ainsi, on peut avoir :

D(int a, int b, int c) : B(a, b, c), A(a, b)

Bien entendu, il sera inutile de préciser des informations pour 'A' au niveau des constructeurs 'B' et 'C'.

En ce qui concerne l'ordre des appels, le constructeur d'une classe virtuelle est toujours appelé avant les autres. Dans notre cas, on a l'ordre 'A', 'B', 'C' et 'D'.

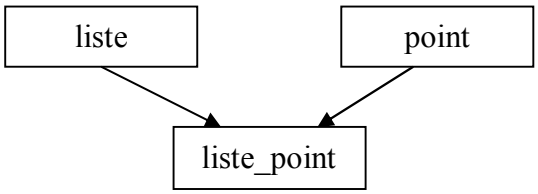
Exemple



L'ordre des appels des constructeurs est : B, A, C, D et E.

Exercice

Interpréter et commenter le programme utilisant l'héritage multiple dans l'exemple suivant de liste simplement chaînée de points (objets appartenant à la classe 'point').



```
struct    element
{
     element   *suivant;
     void      *contenu;
};
//-----------------------------------------
class liste
{
       element *debut;
       element *courant;
     public:
       liste()
       {
          debut=NULL;
          courant=debut;
       }
       ~liste();
       void ajouter();
       void *premier()
       {
          courant=debut;
          return(courant->contenu);
       }
       void *prochain()
       {
          if(courant!=NULL) courant=courant->suivant;
          return(courant->contenu);
       }
       int finir()
       {
          return(courant==NULL);
       }
};
```

Solution avec liste simple d'entiers sans objets :

```
typedef struct sm
{
     int  don;
     sm  *suiv;
} MAILLON;
//---------------------
typedef MAILLON *LISTE;
//---------------------------------------
```

```
void Ajouter(LISTE & l, LISTE & p, int n)
{
      if(l==NULL)
      {   l=new MAILLON;
          l->don=n;
          l->suiv=NULL;
          p=l;
      }
      else
      {   p->suiv=new MAILLON;
          p=p->suiv;
          p->don=n;
          p->suiv=NULL;
      }
}
//-----------------
void Lire(LISTE l)
{
   LISTE p=l;
   while(p!=NULL)
   {
     cout<<p->don<<endl;
     p=p->suiv;
   }
}
//----------------------------
main()
{
     LISTE L1=NULL; LISTE C1=NULL;
     Ajouter(L1,C1,33);
     Ajouter(L1,C1,44);
     Ajouter(L1,C1,55);
     Ajouter(L1,C1,66);
     Ajouter(L1,C1,77);
     Lire(L1);
}
```

Solution avec liste simple avec objets :

```
typedef struct sm
{
     sm  *suiv;
     int contenu;
} MAILLON;
//------------------------------
```

```
class liste
{       MAILLON *debut;
        MAILLON *courant;
     public:
        liste(){ debut=NULL; courant=debut; }
        ~liste(){ }
        void Ajouter(int n)
        {
            if(debut==NULL)
            {   debut=new MAILLON;
                debut->contenu=n;
                debut->suiv=NULL;
                courant=debut;
            }
            else
            {   courant->suiv=new MAILLON;
                courant=courant->suiv;
                courant->contenu=n;
                courant->suiv=NULL;
            }
        }
        void Afficher()
        {
            MAILLON *courant=debut;
            while(courant!=NULL)
            {
             cout<<courant->contenu<<endl;
             courant=courant->suiv;
            }
        }
};
main()
{
  liste l1;
  l1.Ajouter(33); l1.Ajouter(44); l1.Ajouter(55);
  l1.Afficher();
  cout<<endl;
  liste l2;
  l2.Ajouter(333); l2.Ajouter(444); l2.Ajouter(555);
  l2.Afficher();
}
```

Exécution