Fiche C++ n° 4

Principales notions abordées :
- **L'héritage simple**
- **L'héritage multiple**
- **Classes virtuelles**

L'héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d'une classe déjà existante, **la classe de base** (ou super classe).

"Il est plus facile de modifier que de réinventer"

La nouvelle classe (ou **classe dérivée** ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et ainsi peut ainsi réutiliser le code déjà écrit pour la classe de base.
On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

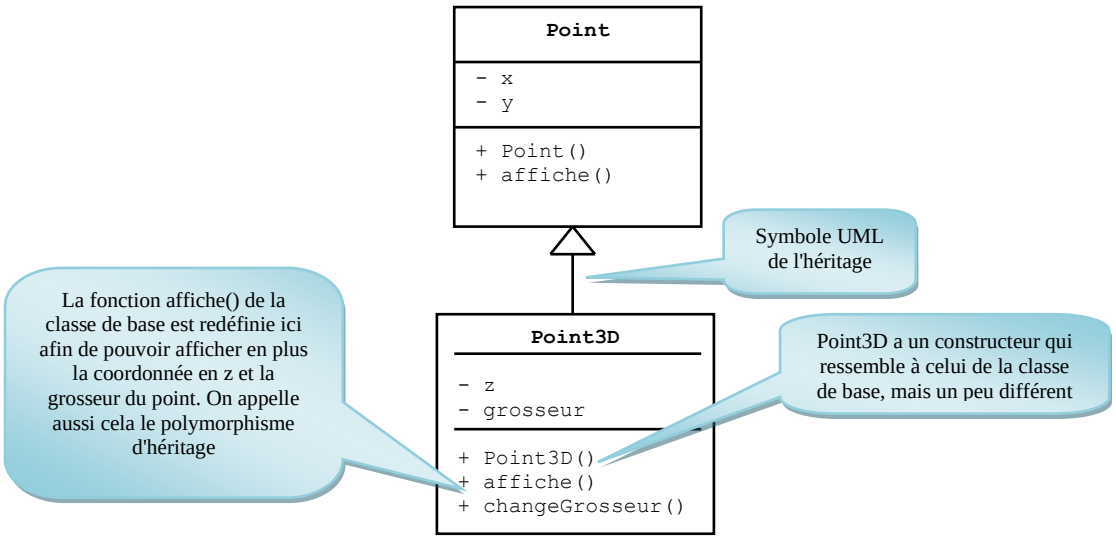
1. Analyse d'un premier exemple

Il s'agit de définir une classe Point3D permettant de définir un point dans l'espace. Cette classe aura en plus une caractéristique qui permettra de définir la grosseur du point.

On peut constater naturellement qu'un point dans l'espace a une coordonnée supplémentaire correspondant à sa cote en z. La grosseur quant à elle correspondra aussi à une donnée membre complémentaire de type entier.

5 fichiers sont mis en œuvre ici.
- point.h : fichier de **déclaration** de la classe Point
- point.cpp : fichier de **définition** des fonctions membres de la classe Point
- point3D.h : fichier de **déclaration** de la classe Point3D
- point3D.cpp : fichier de **définition** de la classe Point3D
- fiche 4_1.cpp : fichier contenant la fonction principale

On peut dire qu'un **point en 3D** "**est une sorte** " de **point en 2D** amélioré. La modélisation UML qui correspond à une généralisation (héritage) est la suivante :



```
//******* point.h ***********
class Point
{
public:
        void affiche();
        Point(float abs=0,float ord=0);

private:
        float y;
        float x;
};
```

```
//********* point.cpp ***********
#include "point.h"

//***********************************
Point::Point(float abs , float ord)
{
        x=abs ; y=ord;
}

//***********************************
void Point::affiche()
{
        cout <<"\nx= "<<x<<" y="<<y;
}
```

Déclaration d'une nouvelle classe Point3D qui hérite en mode public de la classe Point

Mode de dérivation = **public**
Il existe également le mode private et protected. Il ne faut pas confondre le mode de dérivation et la nature des données ou méthodes de la classe

La méthode affiche est redéfinie dans la classe dérivée. En effet, elle existe déjà dans la classe de base. Le système appellera la méthode correspondant à l'objet concerné ; on parle de polymorphisme d'héritage

Nouvelle méthode propre à la classe Point3D

Nouveaux attributs qui viennent s'ajouter à x et à y pour cette classe Point3D

Pour définir un Point3D, il faut fournir x, y ,z et la grosseur du point. On donne à toutes ces données des valeurs par défaut.

```
//****************** point3D.h *******
#include "point.h"

class Point3D : public Point
{
public:
        void changeGrosseur(int NewGrosseur);

        void affiche(void);
        Point3D(float xi=0 , float yi=0 , float zi=0 , int epaisseur=1);


private:
        float z;
        int grosseur;
};
```

```
//********************point3D.cpp **********************
#include "point3D.h"
//************************
Point3D::Point3D(float xi , float yi , float zi , int epaisseur) : Point(xi ,yi)
{
        z=zi;
        grosseur=epaisseur;
}


//*************************
void Point3D::affiche()
{
        Point::affiche();
        cout <<" z= "<<z;
        cout <<" Grosseur ="<<grosseur;
}

//*************************
void Point3D::changeGrosseur(int NewGrosseur)
{
        grosseur=NewGrosseur;
}
```

On complète le constructeur en initialisant la cote en z et la grosseur du point

Appel de la méthode affiche de la classe Point

Pour définir le constructeur d'un Point3D, on commence par appeler le constructeur d'un point en 2D.

Attention, c'est bien un appel du constructeur de Point, pas une déclaration !

```
//********* fiche 4_1.cpp *******

#include "point3D.h"

int main(int argc, char* argv[])
{
        Point   p1(2,4);
        Point3D p2(5,7,11,3);

        p1.affiche();
        p2.affiche();
        p2.changeGrosseur(2);
        p2.affiche();
        return 0;
}
```

La méthode affiche() de la classe Point est appelée car p1 est une instance de la classe Point

2 initialisations pour la classe Point et 4 initialisations pour la classe Point3D

La méthode affiche() de la classe Point3D est appelée car p2 est une instance de la classe Point3D

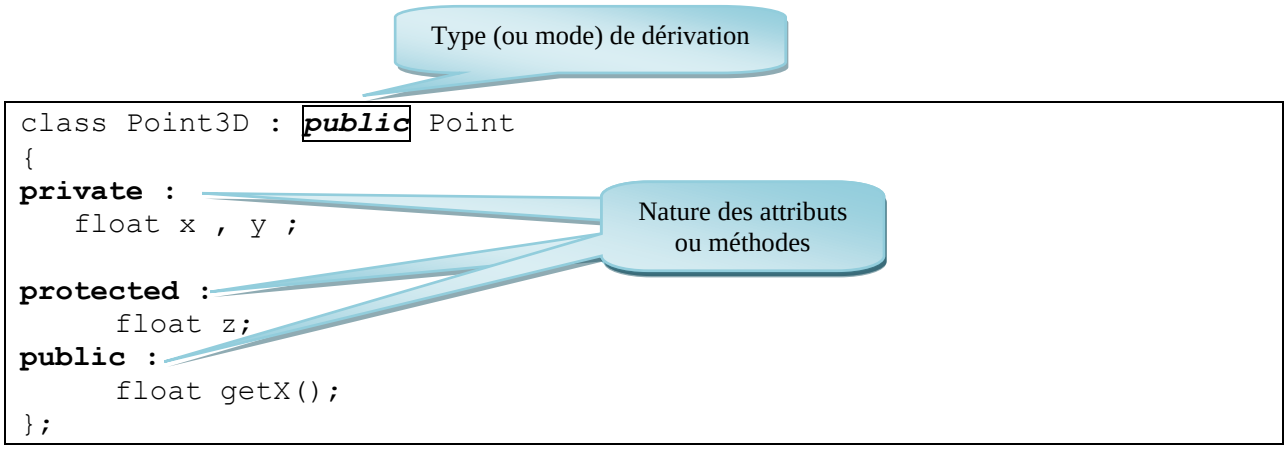
Exercice 1 :

Créez une classe Point3DColore, dérivée de la classe Point3D disposant une donnée membre supplémentaire qui est la couleur du point (Entier Long non signé). Vous prévoirez une nouvelle fonction d'affichage pour ces types de points ainsi qu'un nouveau constructeur. Par défaut, les coordonnées d'un point coloré seront 1,2,5, sa grosseur 3 et sa couleur 128. Vous complèterez aussi le programme principal en créant un objet de ce type et en l'affichant. Par la suite, vous modifierez le programme principal pour ne travailler qu'avec des allocations dynamiques.

2. Les différents modes d'héritage

La structure hiérarchique des classes en C++ favorise les développements séparés. On distingue ainsi le programmeur qui élabore une classe de base (concepteur de la classe de base) et le programmeur qui utilise cette classe pour élaborer d'autres classes dérivées (concepteur des classes dérivées), et le programmeur qui ne fait qu'utiliser les classes (Utilisateur des classes). C'est pour cela que différents modes d'héritage existent.

Reprenons l'exemple de la classe Point3D



Le terme public spécifie le mode de dérivation. Il peut être remplacé par ***private*** ou ***protected***
Ces trois types de dérivation existent pour favoriser ou limiter l'accès des attributs ou méthodes d'une classe de base à ses sous-classes.

Il ne faut pas confondre le type de dérivation avec la nature des attributs ou des méthodes. Pour l'instant, vous connaissez seulement les membres privés(private) et les membres publiques (public). Il convient maintenant de rajouter une troisième possibilité : les membres protégés (protected)**.**

Voici un tableau qui résume les diférents types d'héritage

Mode de dérivation		Statut du membre dans la classe de base (attribut ou méthode)	Statut du membre dans la classe dérivée
private	class B : private A	private	**inaccessible**
		protected	**private**
		public	**private**
protected	class B : protected A	private	**inaccessible**
		protected	protected
		public	**protected**
public	class B : public A	private	**inaccessible**
		protected	protected
		public	public

Mode le plus utilisé

3. Héritage des constructeurs, destructeurs

Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.

Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une liste d'initialisation

L'appel des destructeurs se fait dans l'ordre inverse de celui des constructeurs.

1er cas : les constructeurs n'ont pas de paramètres.

```
//**********point.h **********
class Point
{
public:
        Point();
        ~Point();
};
```

```
//********point3D.h******
#include "point.h"

class Point3D : public Point
{
public:
        Point3D();
        ~Point3D();
};
```

```
//*********fiche 4_3.cpp *******
#include "point3D.h"

int main(int argc, char* argv[])
{
        Point3D *p;
        p=new Point3D;

        //.....

        delete p;
        return 0;
}
```

```
//**********point.cpp ********
#include "point.h"

//*****************************
Point::Point()
{
cout <<"\nCreation  point 2D ";
}

//*****************************
Point::~Point()
{
cout <<"\nDestruction  point 2D ";
}
```

```
//********** point3D.cpp **********
#include "point3D.h"

//************************
Point3D::Point3D()
{
cout <<"\nCREATION POINT  3D";
}


//*************************
Point3D::~Point3D()
{
cout <<"\nDESTRUCTION POINT 3D ";
}
```

Affichage du programme

```
Creation  point 2D
CREATION POINT  3D

DESTRUCTION POINT 3D
Destruction  point 2D
```

2ème cas : Exemple d'appel des constructeurs avec paramètres

Cette façon de faire correspond à celle adoptée dans le premier exemple de cette fiche. Pour construire un point3D, on appelait le constructeur de point en lui passant les coordonnées x et y.

```
//**** extrait de point.cpp **************
Point::Point(float abs , float ord)
{
x=abs ; y=ord;
}
```

Lors de la construction d'un Point3D, **xi est copié dans abs puis yi est copié dans ord** ; le constructeur de Point peut alors initialiser x et y

```
//***** extrait de point3D.cpp ***********************
Point3D::Point3D(float xi , float yi , float zi , int epaisseur) : Point(xi ,yi)
{
z=zi;
grosseur=epaisseur;
}
```

Après l'appel du constructeur de point, l'initialisation du Point3D se termine ici

4. Héritage multiple

L'héritage multiple est possible en C++. Il permet de créer des classes dérivées à partir de plusieurs classes de base.

Dans l'exemple ci dessous, la classe **Pointcolore** hérite simultanément des classes **Point** et **Coul**

```
//*********** coul.h ******************

const int TAILLE_MAX=20;
class Coul
{
public:
        void affiche();
        Coul(char *nom , unsigned long id);

private:
        char NomCouleur[TAILLE_MAX+1];
        unsigned long IdCouleur;
};
```

```
//************** point.h *****************
class Point
{
public:
        void deplace (float tx , float ty);
        void affiche();
        Point(float abscisse=0 , float ordonnee=0);

private:
        float y;
        float x;
};
```

```
//******************** pointcolore.h *****************
#include "point.h"
#include "coul.h"

class Pointcolore : public Point, public Coul
{
public:
        void affiche();
        Pointcolore(float abs=1,float ord=1,char * couleur="noir" , unsigned long id=0x000000);

};
```

Chaque dérivation, ici public pourrait être privée ou protected. L'ordre indiqué (Point puis Coul) correspond à l'ordre d'appel des constructeurs (et à l'ordre inverse de l'appel des destructeurs).

```
//************************** pointcolore.cpp*******************************************
#include "pointcolore.h"

Pointcolore::Pointcolore(float abs,float ord,char * couleur,unsigned long id): coul(couleur,id),Point(abs,ord)
 {
         cout <<"\n\nCreation d'un point colore ";
  }

void Pointcolore::affiche()
{
        Coul::affiche();
        Point::affiche();
}
```

Appel du constructeur de **coul** et de **Point**

L'opérateur de résolution de portée (::) peut être utilisé

- soit lorsqu'on veut accéder à un membre d'une des classes de base, alors qu'il est redéfini dans la classe dérivée
- soit lorsque deux classes de base possèdent un membre de même nom et qu'il faut préciser celui qui nous intéresse.

```
//*********** coul .cpp **************
#include "coul.h"

Coul::Coul(char *nom , unsigned long id)
{
   cout <<"\n\nDefinition d'une couleur ";
   IdCouleur=id;
   strcpy(NomCouleur , nom);
}

void Coul::affiche()
{
   cout <<"\nCouleur "<<NomCouleur <<" IdCouleur="
        <<IdCouleur;
}
```

```
//*********** point .cpp **************
#include "point.h"

Point::Point(float abscisse, float ordonnee)
{
      cout <<"\n\nCreation d'un point ";
      x=abscisse;
      y=ordonnee;
}

void Point::affiche()
{
      cout <<"\nPoint d'abscisse "<<x<<" et d'ordonnee "<<y;
}
```

```
//***************** fiche 4_4.cpp ************************
#include "point.h"
#include "coul.h"
#include "pointcolore.h"

int main(int argc, char* argv[])
{
        Point p1(2,7);
        p1.affiche();

        Coul couleur1("rouge",0xff0000);
        couleur1.affiche();

        Pointcolore p2(3,5,"vert",0x00ff00);
        p2.affiche();

        Pointcolore p3;
        p3.affiche();
        p3.Coul::affiche();
        p3.Point::affiche();

        return 0;
}
```

Définition d'un premier Pointcolore. Regardez bien en observant le résultat de ce programme comment est appelé le constructeur

Utilisation des arguments par défaut

Appel de la fonction affiche de la classe Coul puis de celle de la classe Point en utilisant l'opérateur de résolution de portée (::)

Les constructeurs sont appelés dans l'ordre d'apparition dans la déclaration de l'héritage, et non dans l'ordre des appels aux constructeurs.

5. Méthodes (ou fonctions) virtuelles

Afin de voir l'intérêt des méthodes virtuelles, observons le programme suivant

```
//**** point.h *******
class Point
{
public:
        void affiche();
        Point();
        ~Point();
};
```

```
//***** point3D.h *******
#include "point.h"

class Point3D : public Point
{
public:
        void affiche();
        Point3D();
        ~Point3D();
};
```

```
//*** point.cpp: ****
#include "point.h"

//**********************
Point::Point()
{
      cout <<"\nCreation d'un point ";
}

//********************
Point::~Point()
{
      cout <<"\nDestruction d'un point ";
}

void Point::affiche()
{
      cout <<"\nJe suis un point 2D ";
}
```

```
//***** point3D.cpp ********
#include "point3D.h"

//************************
Point3D::Point3D()
{
      cout <<"\nCREATION POINT EN 3D";
}


//*************************
Point3D::~Point3D()
{
      cout <<"\nDESTRUCTION POINT EN 3D ";
}


void Point3D::affiche()
{
      cout <<"\nJe suis un point 3D ";
}
```

```
#include "point.h"
#include "point3D.h"

int main(int argc, char* argv[])
{
      Point *MonPoint;
      Point3D *AutrePoint;

      AutrePoint = new Point3D;

      MonPoint = new Point;
      MonPoint->affiche();

      MonPoint = AutrePoint;
      MonPoint->affiche();

      return 0;

}
```

Résultats de ce programme

Conclusion :

On peut remédier à ce problème en définissant des méthodes virtuelles. Lorsqu'une méthode est déclarée virtuelle (mot clé virtual) dans une classe, les appels à une telle méthode ou à n'importe laquelle de ses définitions dans des classes dérivées sont résolus au moment de l'exécution, en fonction du type de l'objet concerné.

Ex :

```
class A                          class B  :  public A
   {                                {
   ...                               ....
   public :                         public
      virtual void fct(...);           void fct(...);
      ...                              ...
   };                               };
```

Règles :

- Le mot clé virtual ne s'emploie qu'une fois pour une fonction donnée. il ne doit donc pas accompagner la redéfinition de cette méthode dans les classes dérivées.
- Une méthode déclarée virtuelle dans une classe de base peut ne pas être redéfinie dans ses classes dérivées.
- Une méthode virtuelle peut être redéfinie
- Un constructeur ne peut pas être virtuel, un destructeur peut l'être.

Corrigez le programme pour que le problème constaté ne se produise plus puis suivez le tutoriel vidéo que j'ai créé pour vous montrant l'intérêt des méthodes virtuelles en environnement graphique.