TP3
SLAM2
Héritage

Ce travail nécessite une lecture des fondamentaux C++, pour mieux comprendre les concepts de la POO en générale.
Pourquoi ?

- Parce que le langage C++ est celui qui utilise le moins d'artifice pour masquer la réalité.
- Parce que tous les nouveaux langages empruntent en partie tout l'héritage de C++. Comprendre C++ vous donnera donc un bagage utile, même pour faire du PHP.

Le travail demandé ici, peut se réaliser avec le langage que vous souhaitez, et l'IDE que vous souhaitez.

L'héritage à mettre en ouvre est simple. Il est issu de l'exemple suivant :

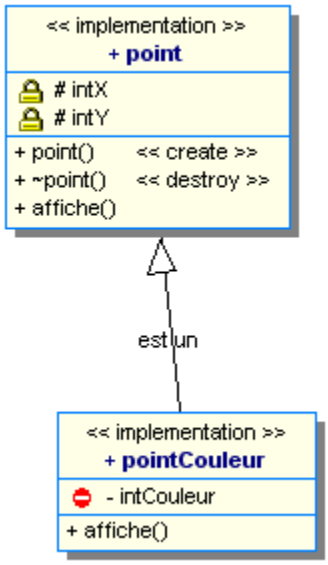
Prenons l'exemple de la classe point :

La classe point possède deux membres privés intX et intY. Elle possède des méthodes publiques comme son constructeur, son destructeur et une méthode d'affichage des données du point.

La classe pointCouleur est dérivée de la classe point. Sa spécificité est d'associer une couleur à un point. Elle définie donc un membre privée couleur.

En modifiant l'accessibilité aux membres des deux classes proposez un DCL faisant apparaître l'héritage.

Le DCL est le suivant :



1. L'héritage simple

Le DCL suivant nous permet d'introduire l'héritage à partir d'une classe

1.1. Notation

La dérivation introduit simplement la notion suivante dans l'interface :

class B : <type du contrôle d'accès> A // B dérive de A

Le type du contrôle d'accès peut être, public, private, protected.

1.2. Exemple

Ecrire l'interface et l'implémentation issue du DCL

```
#ifndef _POINT_H
#define _POINT_H


#include <iostream>
using namespace std;

class point
{
	protected :

		int intX;
		int intY;

	public :

		point(int intX = 0, int intY = 0);
		friend ostream & operator << (ostream & out, point &a);
		//virtual void affiche(); // ligature dynamique de l'appel de
affiche suivant l'objet qui l'utilise (polymorphisme)
		void affiche();
		~point();
};
#endif _POINT_H
```

```
#ifndef _POINT_CPP
#define _POINT_CPP

#include "point.h"

point::point(int intX, int intY)
{
	this->intX = intX;
	this->intY = intY;
}

void point::affiche()
{
```

```
	cout<<" mon X = "<<this->intX<<" mon Y = "<<this->intY<<"\n";
}

ostream & operator <<(ostream & out, point &a)
{
	out<<" X = "<<a.intX<<" Y = "<<a.intY<<"\n";
	return out;
}

point::~point()
{
}
#endif _POINT_CPP
```

```
#ifndef _POINTCOULEUR_H
#define _POINTCOULEUR_H

#include "point.h"

class pointCouleur : public point
{
	private :

		int intCouleur;

	public :

		pointCouleur(int intX = 0, int intY = 0, int intCouleur = 0) ;
		friend ostream & operator << (ostream & out , pointCouleur &a);
		void affiche();
		int getCouleur(); // accesseur ajouté pour pouvoir afficher la
couleur dans le cout redéfini
		~pointCouleur();



};
#endif _POINTCOULEUR_H
```

```
#ifndef _POINTCOULEUR_CPP
#define _POINTCOULEUR_CPP


#include "pointCouleur.h"

void affiche()
{
	cout<<"hello \n";
}

pointCouleur::pointCouleur(int intX, int intY, int intCouleur):point(intX,
intY)
{
	this->intCouleur = intCouleur;
}

void pointCouleur::affiche()
{
	cout<<"La couleur = "<<this->intCouleur<<" X = "<<this->intX<<" Y =
"<<this->intY<<"\n"; // OK mais on appeler affiche de point
```

```
	//point::affiche(); //OK appel la fonction affiche de point supprimer
alors l'affichage précédent de X et Y dans le cout avec la couleur
	//this->affiche(); // recursivité sans fin OK
	//affiche(); // recursivité sans fin OK
	//::affiche(); // appel de affiche procédure globale OK
}

int pointCouleur::getCouleur()
{
	return(intCouleur);
}

ostream & operator <<(ostream & out, pointCouleur &a)
{
	//out<<" couleur = "<<a.intCouleur; // illegal call création d'un
objet a qui ne peut pas manipuler directement la couleur
	//out<<" couleur = "<<a.intX; // illegal cannot acces private member
	//Il faut ici absolument un accesseur pour accéder à la couleur de
même si on veut directement X et Y
	out<<" couleur = "<<a.getCouleur();
	// au lieu de faire des accesseur sur X et Y , on peut réutiliser
affiche de point et l'appeler ainsi :
	a.point::affiche();


	return out;
}

pointCouleur::~pointCouleur()
{
}


#endif _POINTCOULEUR_CPP
```

1.3. Conséquence de l'héritage

L'héritage implique certaines choses :

Remarque à propos de la construction et de la destruction.

Dans le cas de l'héritage simple, les constructeurs sont exécutés en partant de la racine de l'arbre d'héritage et en descendant vers les classes dérivées. Pour le destructeur le mécanisme fonctionne en sens inverse.

Les constructeurs et les destructeurs ne sont par transmis pas héritage. S'ils ne sont pas mentionnés dans la classe dérivée, seuls les constructeurs et destructeurs triviaux sont valables.

Remarque sur le clonage :

Le sens du clonage des objets, soit de la classe dérivée vers la classe de base, provient d'une compatibilité entre objets d'une classe de base et classe dérivée. Elle revient à convertir l'objet de la classe dérivée dans le type de base (c'est-à-dire à ne considérer de l'objet de la classe dérivée que ce qui est du type de la classe de base) et à affecter ce résultat au clone de la classe de base.

```
pointCouleur B ;
point D = point(B) ; // OK constructeur de recopie synthétisé ou trivial
```

Remarque sur la redéfinition des membres :

Il est possible de redéfinir un membre dans une classe dérivée (affiche dans notre cas). Le membre redéfini masque le membre issu de la classe de base. L'opérateur de résolution de portée :: permet cependant d'accéder au membre caché.

```
Void affiche()
{
        Cout<<''hello'' ;
}

Void pointCouleur ::affiche()
{
        Cout<<intCouleur ;
        Point ::affiche() ; // appel d'affiche() de la classe point
        Affiche() ; // appel d'affiche de pointCouleur (récursivité)
        This->affiche() ; // appel d'affiche de pointCouleur (récursivité)
        ::affiche() ; // appel d'affiche la procédure non déclarée dans la classe
}
```

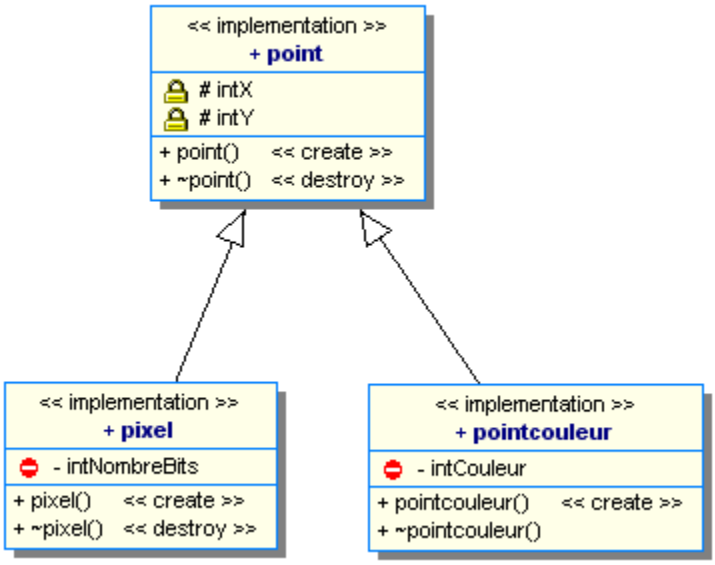
2. L'héritage multiple

Deux visions sont possibles :

- Une classe de base à plusieurs classes dérivées
- Une classe de base à plusieurs enfants qui ont le même enfant

- Une classe point possède deux classes dérivées pointcouleur et pixel.
- Une classe point possède deux classes dérivées pointcouleur et pixel, qui ont toutes les deux la même classe dérivée photophore

Dans le premier cas, rien ne change par rapport à l'héritage simple.
Dans le deuxième cas, il faut simplement ajouter par quel côté va se faire l'héritage. Dans ce cas ni par la gauche, ni par la droite, simplement de manière directe grâce au mot clé « virtual »



Dans ce cas pas de changement notables par rapport à l'héritage simple

Ajouter dans votre code la classe pixel pour quelle puisse être utilisée dans votre application.