L'HERITAGE EN C++.

1. RAPPEL.

la programmation orientée objet : P.O.O.

Ce type de programmation (opposé à la programmation procédurale) est basé sur l'écriture de classes. Une classe contient ses membres : les données et les méthodes (fonctionnalité) nécessaires au traitement des données. Chaque membre possède un type d'accès : **privé**, **protégé** ou **public**.

Ainsi, avant même de prévoir le déroulement du programme à réaliser, il faut étudier chaque élément intervenant dans le processus et créer sa classe qui définit les grandeurs mises en jeu (données), leur traitement (méthode) et les droits d'intervention sur les membres.

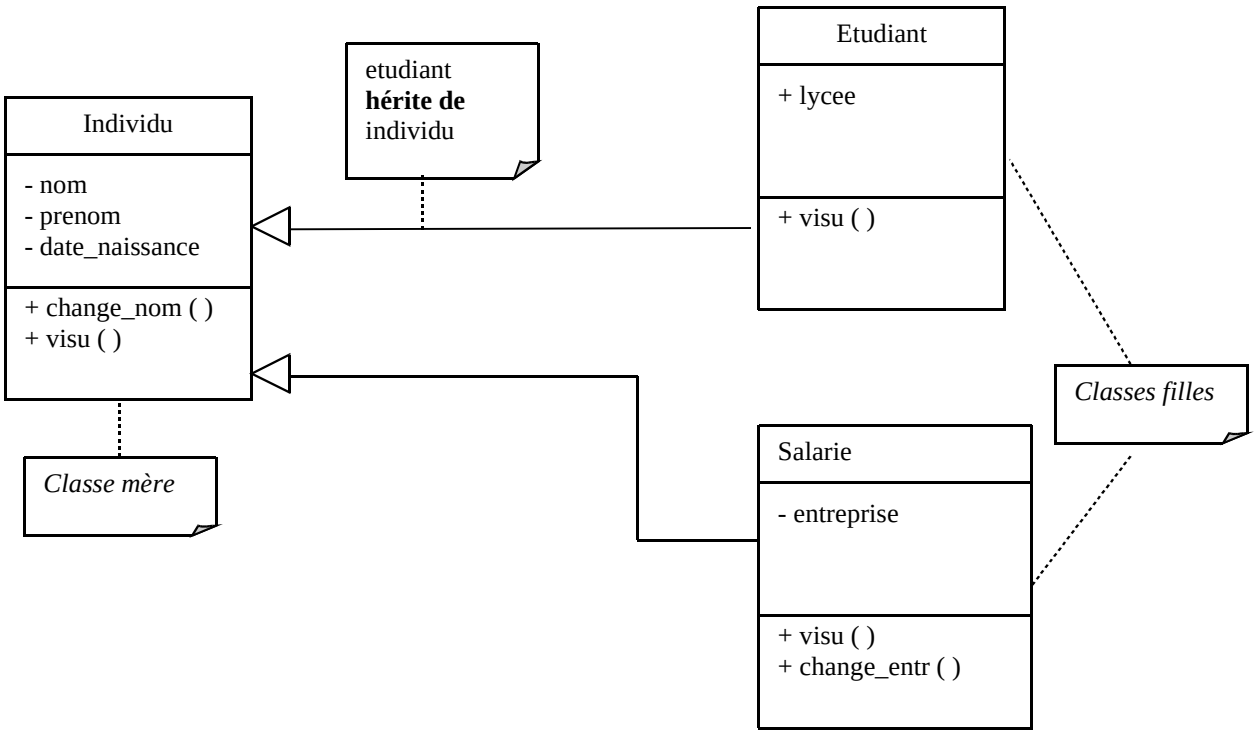
La programmation consiste ensuite à envoyer des messages aux objets pour leur demander d'exécuter une de leurs méthodes.

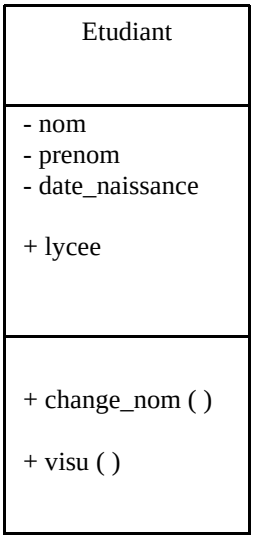
L'écriture d'une classe est généralement protégée : la modification des membres d'une classe n'est pas souhaitable.
Par contre, pour enrichir une classe déjà écrite, on crée des classes dérivées.

2. PRINCIPE.

L'héritage est un mécanisme qui permet à une classe de dériver d'une autre et donc de posséder les mêmes données et méthodes que la classe de base, plus de nouvelles données et méthodes définies dans la nouvelle classe. On parle dans ce cas de « **spécialisation** » ou de « **généralisation** ».
Ex :
classe **etudiant** hérite de classe **individu**

Représentation UML (Unified Modeling Language) d'un héritage simple :





Quelle différence existe-t-il entre cette représentation UML et celle de dessus ?

Un objet de type « **Etudiant** » hérite des données et fonctions membres de la classe « **Individu** » dont il dérive.
Un objet de type « Etudiant » possédera donc :

Données membres :

nom
prenom
date de naissance
lycee

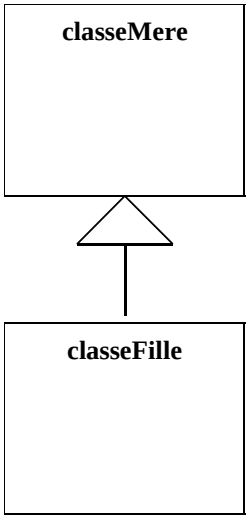
Fonctions membres :

change_nom
visu (celle définie dans « individu »)
visu (celle définie dans « etudiant »)

Les deux fonctions « **visu** » sont différentes.
RMQ : Constructeurs et destructeurs ne sont pas représentés mais existent.

3. UN EXEMPLE.

L'héritage en UML est représenté par un dessin :



Sa traduction en C++ est

class etudiant **: public** individu

```
/*fichier exemple d'héritage corps des méthodes et programme principal*/
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <alloc.h>
#include "herit.h"
//----classe individu
individu::individu(char *init_nom,char *init_prenom,char * init_naiss)
{
nom = (char *)malloc(strlen(init_nom));
nom = init_nom;
prenom = (char *)malloc(strlen(init_prenom));
prenom = init_prenom;
naiss = (char *)malloc(strlen(init_naiss));
naiss = init_naiss;
}
void individu::change_nom()
{
char *n = (char *)malloc(100);
cout << "\nentrez le nouveau nom de la personne : "<<nom<<"\n";
cin >>n;
nom = (char *)malloc(strlen(n)+1);
nom = n;
}
void individu::visu()
{
cout <<"\nnom : "<<nom << "\tprenom : "<<prenom << "\tdate de naissance : "<<naiss;
}
//fin classe individu
//classe etudiant
etudiant::etudiant(char * init_lycee,char *init_nom,char *init_prenom,char * init_naiss)
:individu(init_nom,init_prenom,init_naiss)
{
lycee = (char *)malloc(strlen(init_lycee));
lycee = init_lycee;
}
void etudiant::visu()
{
//visu();//interdit car ambigue
//this->individu::visu();
individu::visu();
cout << "\nlycee : "<<lycee;
}
//fin de classe etudiant
```

```
int main()
{
char *fin = (char *)malloc(5);
individu i1("trucnom","trucprenom","1975");
etudiant e1("enrea","monnom","monprenom","1950") ;
individu * pti1;
//pte1=&i1; //interdit, pas de conversion possible
pti1 = &e1;
pti1->visu(); //fonction "visu" de individu appelée
i1.visu();
e1.visu();
e1.change_nom();
e1.visu();
e1.individu::visu();
        return 0;
}
```

4. MISE EN ŒUVRE.

La mise en œuvre est on ne peut plus simple, il suffit d'écrire dans le fichier de déclaration de la classe fille:

class classe_fille **:** mode_d'héritage(**private**, **protected**, **public**) classe_mere

Le mode d'héritage permet de préciser les droits d'accès aux données de la classe mère dans la classe dérivée ou dans le programme écrit par l'utilisateur

class Etudiant : public Individu //mode de dérivation publique

Le reste est identique à l'écriture d'une classe.

Question :
Dessiner au format UML la représentation de la classe **lecteurRFID** héritant de la classe **rs232**.

5. UTILISATION DES DONNEES MEMBRES ET METHODES DE LA CLASSE MERE DANS LA CLASSE FILLE.

On peut résumé les choses de la façon suivantes dans le cas de la **dérivation publique**:

Déclaration dans la classe mère	Private	Protected	Public
Droits d'accès dans la classe fille	**NON**	**OUI**	**OUI**
Droit d'accès dans le programme	**NON**	**NON**	**OUI**

```
void etudiant::visu()
{
//cout << nom ;                    //interdit car la donnée est « private ».
individu::visu();
cout << "\nlycee : "<<lycee;
}
```

Ce tableau s'applique aussi aux méthodes :

6. REDEFINITION DES METHODES.

Il est possible dans une classe dérivée de créer une nouvelle fonction possédant le même nom qu'une fonction de la classe mère. Pour cela, afin de lever toute ambiguïté dans l'utilisation de ces fonctions, utiliser l'opérateur de résolution de portée.

```
void etudiant::visu()
{
//visu();//interdit car ambiguë et potentiellement récursif
//this->individu::visu();
individu::visu();                  //appel à « visu » de la classe mére

int main()
{
etudiant e1("enrea","monnom","monprenom","1950") ;
e1.visu();
e1.change_nom();
e1.visu();                         //appel à « visu » de la classe fille car appelée par e1.
```

7. PASSAGE DES ARGUMENTS A LA CLASSE MERE.

Lorsque de l'instanciation d'un objet d'une classe dérivée, le système appelle en premier le constructeur de la classe mère (=> passage d'argument au constructeur) puis en second le constructeur de la classe fille.

Les arguments destinés aux constructeurs de la classe mère sont définis lors de la **définition** des constructeurs de la classe fille et non lors de sa déclaration. Le passage d'arguments se fait comme suit :

```
cl_fille : :cl_fille ( int  gg  ,  char  *  te )   :  cl_mere ( te )
{….
}
```

Dans ce cas la valeur passée en argument « **te** » sera transmise à la classe mère en place et lieu de « **arg** ».
Il s'agit d'un appel au constructeur de la classe mère, ne pas réécrire les types des paramètres.
Il est nécessaire dans ce cas que la classe mère possède un constructeur de prototype :
cl_mere (char * arg)

```
Individu   (        char *init_nom,
                    char *init_prenom,
                    char * init_naiss
                     );

etudiant::etudiant ( char * init_lycee, char *init_nom, char *init_prenom, char * init_naiss)
: individu(init_nom,  init_prenom,  init_naiss)
//        : individu(init_nom,  «  monprenom »,  init_naiss)
//        : individu(init_nom,  «  monprenom »,  1956)
//        : individu(init_nom)
//        : individu(init_naiss,  «  monprenom »,  init_nom)

{….
}
int main()
{
etudiant e1("enrea","monnom","monprenom","1950") ;
```

QUESTION :
Dans l'exemple ci-dessus, rayer en justifiant les passages d'arguments incorrects.

8. CONTROLE D'ACCES.

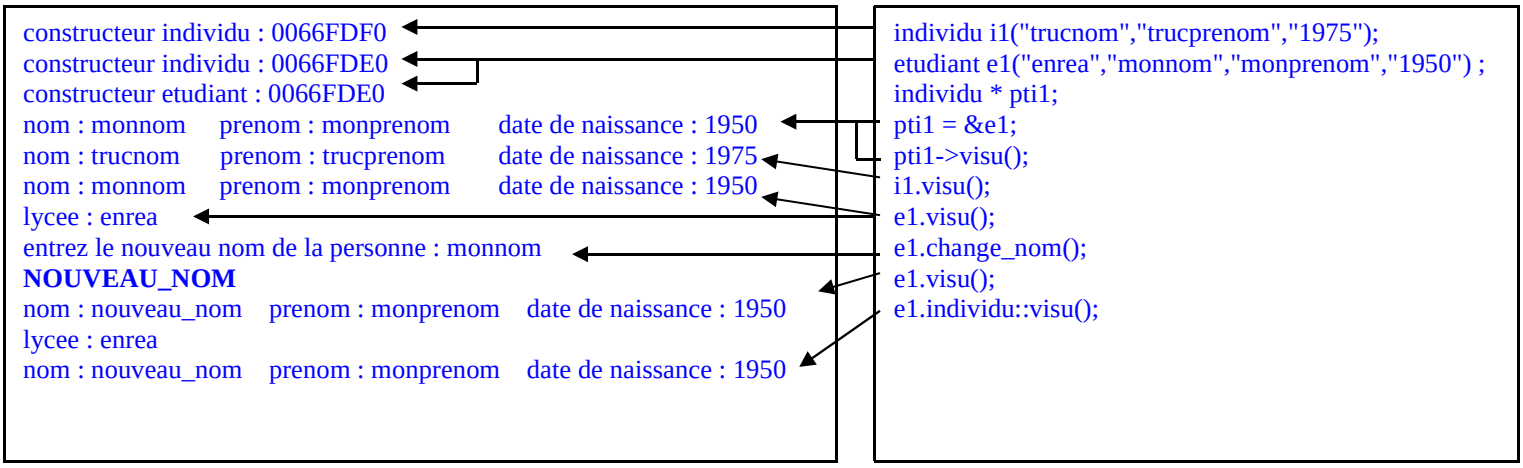
Le contrôle d'accès est défini lors de la déclaration.

```
class truc   :   private    machin
```

Type de dérivation	Déclaration dans la classe de base	Accès dans la classe dérivée
public	Public	Public
	Protégé	Protégé
	Privé	Privé (inaccessible)
Protégé	Public	Protégé ①
	Protégé	Protégé
	Privé	Privé (inaccessible)
Privé	Public	Privé (accessible) ②
	Protégé	Privé (accessible) ③
	Privé	Privé (inaccessible)

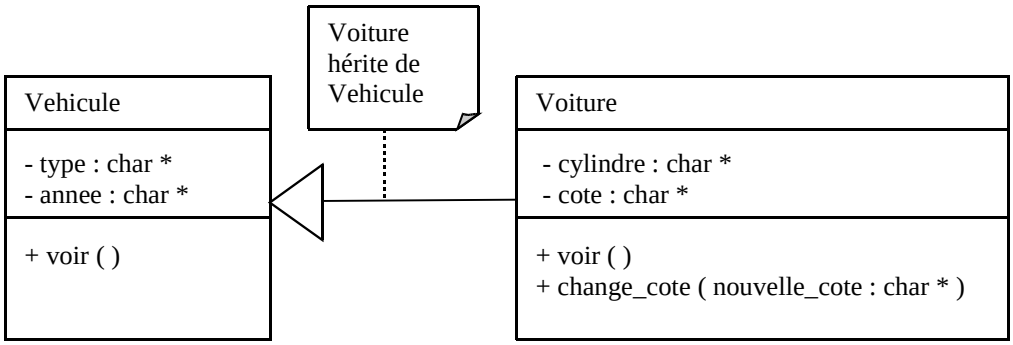
Remarque : les membres déclarés privés dans la classe de base ne sont pas accessibles pour un objet de la classe dérivée.

9. RESULTAT DE L'EXECUTION DE L'EXEMPLE :



TD HERITAGE SIMPLE

L'objectif est d'écrire 2 classes se présentant comme suit :



Les constructeurs ne sont pas représentés.
Le constructeur de la classe « Vehicule » doit initialiser les données membres privées «type » et « annee ».
Le constructeur de la classe « Voiture » doit initialiser les données membres privées « cylindre » et « cote » ainsi que les données membres de la classe mère (passage des arguments de la classe fille vers la classe mère).
Les deux méthodes « voir ( ) », sont différentes et permettent de visualiser les données membres de l'objet .

- Peut-on modifier les données membres de la classe Vehicule dans les méthodes de la classe Voiture ? Justifier.
- Comment utiliser la méthode « voir ( ) » de la classe Vehicule dans la méthode « voir ( ) » de la classe Voiture ?
- Combien de paramètres doit-on passer en arguments aux constructeurs des classes Vehicule et Voiture ?
- Donner les prototypes des constructeurs des classes Vehicule et Voiture.
- Un objet de type Vehicule peut-il accéder à la méthode « voir ( ) » de la classe Voiture ? Justifier.
- Que doit-on écrire dans un programme pour qu'un objet de type voiture puisse accéder à la méthode « voir ( ) » de la classe Vehicule.
- Ecrire les classes Vehicule et Voiture.