TD-TP : *LIENS ENTRE CLASSES*

MISE EN PLACE

1. Deux classes

Etant données la classe `Voiture` dont les objets sont caractérisés par une `marque` et une `plaque` d'immatriculation, **et** la classe `Individu` dont les objets sont caractérisés par un `nom` et un `prénom`, chaque classe ayant un constructeur.

Travail à faire

1.a Ecrire pour chacune de ces 2 classes une méthode `string toString()` qui retourne la description de l'objet concerné,

Et écrire un `main()` qui :

1.b Crée une Voiture `voit1` « `RenaultClio, 123AB64` », une `voit2` « `Peugeot106, 678CD96` », `voit3` « `CitroenPicasso, 456EF75` » et les Individu `ind1` « `Dupond, Pierre` », `ind2` « `Martin, Louis` » et `ind3` « `Durand, Marcel` ».

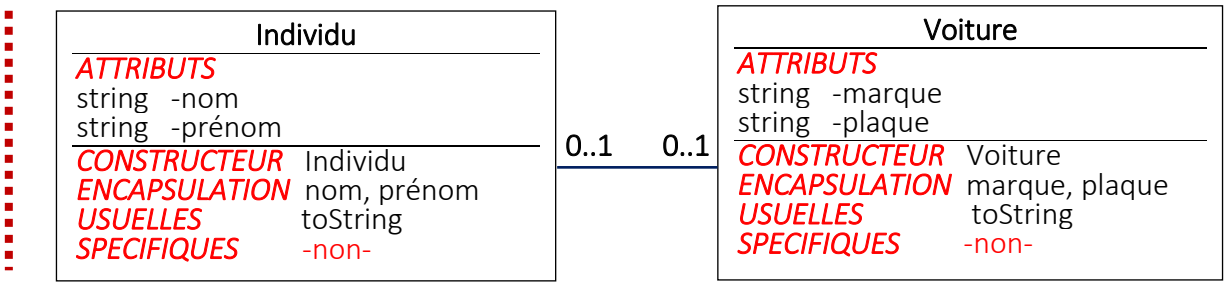
1.c Affiche les attributs de chacun des objets créés en utilisant les méthodes `toString()` respectives.

```
#include "Voiture.h"
#include "Individu.h"

int main(int argc, char *argv[]) {

 // 1. MISE EN PLACE
 cout << "\n\n1 - MISE EN PLACE\n\n";

 // Déclaration des objets
  Voiture  voit1 ("RenaultClio", "123AB64"),
           voit2 ("Peugeot106", "678CD96"),
           voit3 ("CitroenPicasso", "456EF75");
  Individu ind1 ("Dupond","Pierre"),
           ind2 ("Martin", "Louis"),
           ind3 ("Durand", "Marcel");

  // Affichage de leurs valeurs
  cout << ind1.toString() << endl
       << ind2.toString() << endl
       << ind3.toString() << endl;
  cout << voit1.toString() << endl
       << voit2.toString() << endl
       << voit3.toString() << endl;
}
```

RELATION SIMPLE ENTRE DEUX CLASSES

Tout `Individu` peut avoir au plus une `Voiture` et une `Voiture` peut avoir zéro ou un `Individu` comme pilote.

Travail à faire : Représenter cette relation avec un schéma UML.



Note : *Lorsqu'il existe une relation entre deux classes* **`A`** *et* **`B`***, il faut se poser des questions de conception (cf. questions conceptuelles), quant à la* ***navigabilité entre les objets****. A savoir :*

- ***Un objet de* `A` *doit-il connaître l'objet de* `B` *avec lequel il est en relation ?***

et réciproquement

- ***Un objet de* `B` *doit-il connaître l'objet de* `A` *avec lequel il est en relation ?***

Les réponses à ces deux questions guideront quant à l'implémentation la plus adaptée.

☹ *Quelle que soit la réponse à ces questions, en C++* ***une mauvaise implémentation*** *consiste à déclarer dans la classe* **`A`***,* ***un attribut `monPoteB` qui serait du type classe `B`****.*

En effet, l'attribut `monPoteB` *sera initialisé aux mêmes valeurs que celles contenues par l'objet avec qui se lier, mais* `monPoteB` *ne sera pas impacté par les changements de valeurs que subira l'objet lié.*

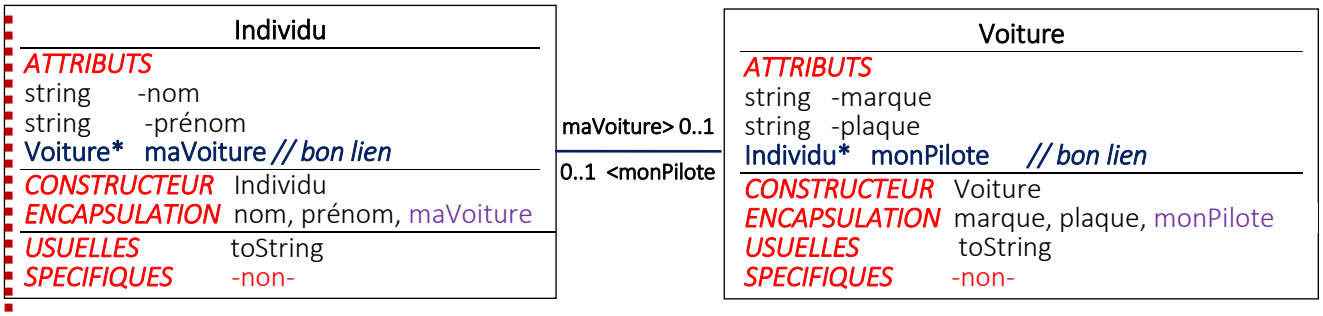
☺ ***Une bonne implémentation*** *consiste à déclarer dans la classe* **`A`***,* ***un attribut `monPoteB` qui serait du type pointeur de `B`****.*

En effet, l'attribut `monPoteB` *sera initialisé par l'adresse de l'objet avec qui se lier,* `monPoteB` *permettra d'accéder à l'objet lié et à ses attributs qui enregistreront les éventuels changements.*

2. Implémenter une bonne relation entre objets

Travail à faire

2.a Modifier le code des classes `Voiture` et `Individu` en intégrant respectivement l'attribut `Individu* monPilote;` dans la classe `Voiture` (cf. Voiture.h) et l'attribut `Voiture* maVoiture;` dans la classe `Individu` (cf.Individu.h).



2.b Coder dans le `main` le fait que la Voiture `voit3` et l'Individu `ind3` sont en relation.

2.c Afficher la plaque d'immatriculation de la voiture que conduit `ind3`.

2.d Changer la plaque d'immatriculation de la `voit3` en lui attribuant la valeur « `77777NO22` ».

2.e Afficher la plaque d'immatriculation de la voiture que conduit `ind3`. Aura-telle changé ?

```
// DANS LE MAIN
// 2. BON LIEN
cout << "\n\n2 - Bon lien\n\n";

// 2.b Pointage réciproque ind3 <=> voit3
ind3.maVoiture = &voit3; voit3.monPilote = &ind3;

// 2.c Affiche la plaque de la voiture pilotée par ind3
cout << "La voiture de " << ind3.toString();
cout << " est " << ind3.maVoiture->toString ();
cout << endl << endl;

// 2.d Change la plaque de la voiture voit3
voit3.imat = "77777NO22";
cout << "La nouvelle plaque de " << voit3.marque
     << " est devenue " << voit3.imat << endl << endl;

// 2.e Constate que la plaque de la voiture a changé
cout << "La voiture de " << ind3.toString();
cout << " est " << ind3.maVoiture->toString();
cout << endl << endl;
```

3. Afficher les attributs de l'objet avec lequel on est en relation

Nous envisageons une méthode string toStringAndLink() pour chacune des deux classes, qui retourne la description complète de l'objet concerné.

Travail à faire

3.a Dans le cas de la classe `Individu`, en plus de la description de l'individu, si ce dernier possède une voiture, toStringAndLink() retourne les attributs de la voiture en question.

```
string Individu::toStringAndLink() {
    string resultat;
    resultat = this->toString();
    if (maVoiture != nullptr) {
          resultat += "Pilote : " + maVoiture->toString();
    }
    return resultat;
}
```

3.b Dans le cas de la classe Voiture, en plus de la description de la voiture, si cette dernière est pilotée par un individu, toStringAndLink() retourne les attributs de l'individu en question.

```
string Voiture::toStringAndLink() {
    string resultat;
    resultat = this->toString();
    if (monPilote != nullptr) {
          resultat+= "A le pilote : " + monPilote->toString();
    }
    return resultat;
}
```

3.c Dans le main afficher la description complète de ind3 et voit3, qui sont liés entre eux, ainsi que la description complète de ind2 et voit2 qui eux, ne sont pas liés.

```
// 3. AFFICHER TOUT
cout << "\n\n3 - Afficher tout - dont l'eventuel correspondant\n\n";
cout << ind3.toStringAndLink() << voit3.toStringAndLink();
cout << ind2.toStringAndLink() << voit2.toStringAndLink();
```

4. Assurer la symétrie de la relation

> *Si une relation entre une `classeA` et une `classeB` est réciproque (cf. navigable dans les 2 sens), on peut considérer que :*
> *1. **dès lors qu'un lien est établi** d'un objet `a` vers un objet `b`,*
> *2. alors **le lien réciproque** de `b` vers `a` **est également à établir***

Dans la version de programme C++ réalisée jusqu'ici, le pointage réciproque automatique n'est pas implémenté.

Si on souhaite implémenter, un lien réciproque entre `a` et `b`, il faut préalablement s'assurer que `a` (respectivement `b`) n'est pas déjà lié avec un objet `x` (respectivement `y`). En effet, si un des objets est lié, il faut préalablement le délier de son correspondant.

Pour cela :

Travail à faire

4.a Ecrire le code la méthode `majMaVoiture(Voiture*)` de la classe `Individu`, qui modifie simplement l'attribut `maVoiture`. De façon symétrique, écrire la méthode `majMonPilote(Pilote*)` de la classe `Voiture`, qui modifie simplement l'attribut `monPilote`.

```
void Individu::majMaVoiture(Voiture* voiture){
        maVoiture = voiture;
}
```

```
void Voiture::majMonPilote(Individu* pilote){
        monPilote = pilote;
}
```

4.b Ecrire le code de la méthode `supprimerLien()` pour un objet de la classe `Voiture` et de façon symétrique, écrire le code de la méthode `supprimerLien()` pour un objet de la classe `Individu`.

```
void Voiture::supprimerLien () {
    if (monPilote != nullptr) { // Si il y a un lien
        monPilote -> majMaVoiture (nullptr) ;
        majMonPilote (nullptr);
    }
}
```

```
void Individu::supprimerLien () {
    if (maVoiture != nullptr) { // Si il y a un lien
        maVoiture -> majMonPilote (nullptr) ;
        majMaVoiture (nullptr);
    }
}
```

4.c Ecrire le code de la méthode `setMaVoiture(Voiture*)` dans la classe `Individu` de sorte que son utilisation délie les objets éventuellement liés et assure l'initialisation du pointeur local ainsi que l'initialisation de son vis-à-vis dans l'autre objet. Idem pour la méthode `setMonPilote(Pilote*)` de la classe `Voiture`.

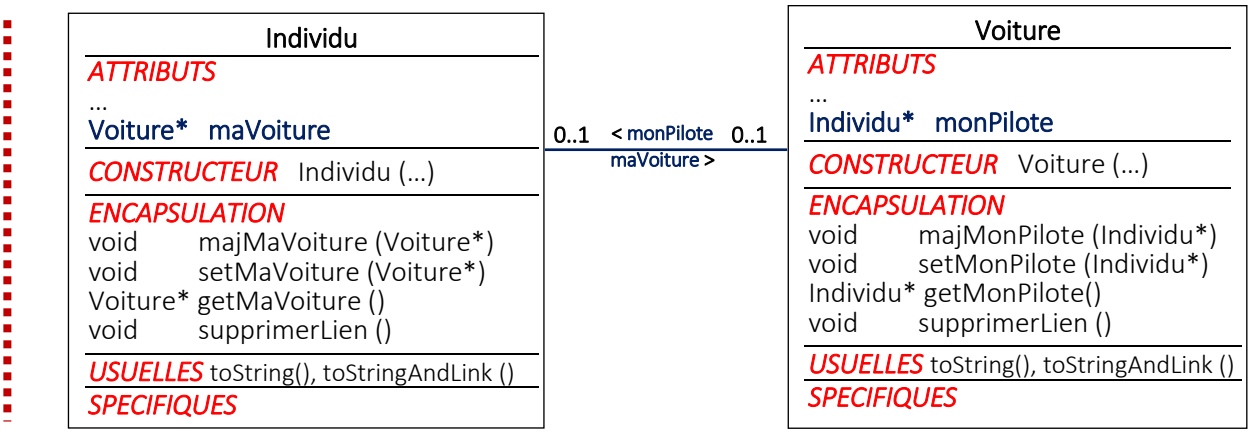
```
// Assure le lien réciproque (idem dans Individu)
void Voiture::setMonPilote(Individu* pilote) {

   // Supprimer mon éventuel lien actuel
   supprimerLien ();

   if (pilote != nullptr) {
      // Supprimer l'éventuel lien actuel de mon correspondant
      pilote -> supprimerLien ();

      // Etablir le lien croisé avec mon correspondant
      majMonPilote(pilote);          // Ie pointe sur lui
      pilote -> majMaVoiture(this);  // Il pointe sur moi
   }
}
```

4.d Dans le `main`, créer le lien entre `ind3` et `voit3` en utilisant `setMaVoiture` de la classe `Individu`, vérifier que les 2 liens sont effectifs. Puis lier `voit3` avec `ind2` en utilisant `setMonPilote` et vérifier que `ind3` n'est plus lié, alors que `voit3` et `ind2` le sont.

```
// 4. LIEN SYMETRIQUE
cout << "\n\n4 - Gestion de la symétrie du lien avec un seul appel !\n";
ind3.setMaVoiture (&voit3);   // Lie ind3 et voit 3 avec un seul 'set'
cout << voit3.toStringAndLink() << endl << ind3.toStringAndLink() << endl << endl;
voit3.setMonPilote (&ind2);   // Lie voit3 à ind2 avec un seul 'set' en déliant ce qui est à délier
cout << ind3.toStringAndLink() << endl << voit3.toStringAndLink() << endl << voit2.toStringAndLink();
```

4.e Un schéma générique pour la réciprocité des liens :

- Faire le schéma UML correspondant aux méthodes codées.



- En quoi les méthodes codées jusqu'à présent s'apparentent au schéma général ci-dessous ?

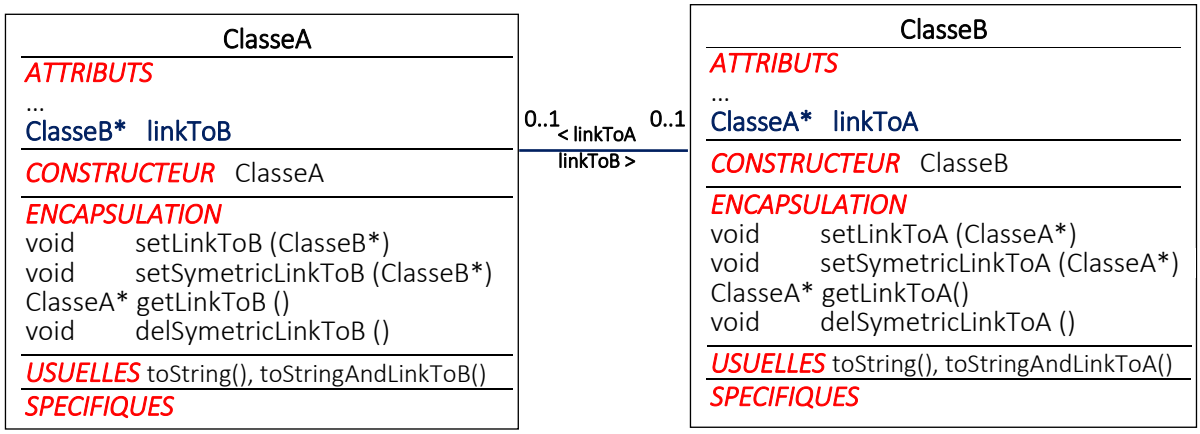


Schéma UML relatif à l'implémentation de liens symétriques

Parmi les éléments de correspondance à identifier :

	ATTRIBUTS	
ClasseA* linkToA	⇔	Voiture* maVoiture
	ENCAPSULATION	
void setLinkToA (ClasseA*)	⇔	void majMaVoiture (maVoiture*)
void setSymetricLinkToA (ClasseA*)	⇔	void setMaVoiture (maVoiture*)
ClasseA* getLinkToA ()	⇔	Voiture* getMaVoiture ()
void delSymetricLinkToA ()	⇔	void supprimerLien ()

Bien distinguer la mise à jour simple de la mise à jour symétrique du lien.

5 –Destruction d'un objet, lorsqu'il est lié

La destruction d'un objet lorsqu'il est lié à un autre objet doit supprimer le lien qui pointe sur lui.

Travail à faire

5.a Ecrire les destructeurs `~Voiture()` et `~Individu()` en conséquence.

```
Voiture::~Voiture() {
   supprimerLien();
}
```

```
Individu::~Individu() {
   supprimerLien();
}
```

5.b Dans le `main` afficher la description complète pour `ind2` et `voit3`, liés entre eux, détruire `voit3`, puis afficher la description complète de `ind2` pour constater que son lien vers `voit3` a été supprimé.

```
// 5. DéTRUIRE OBJET LIé
cout << "\n\n5 - Detruire un objet lie\n\n";
cout << ind2.toStringAndLink(); // Objet ind2 lié à voit3
delete (&voit3);                // Destruction de voit3
cout << ind2.toStringAndLink(); // On observe que l'objet ind2 n'est plus lié
```