POO C++

Sommaire

Initiation à la programmation orientée objet

Notion d'objets

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Concrètement, un **objet** est une **structure de données** (**ses attributs** c.-à-d. des variables) qui définit son **état** et une **interface** (**ses méthodes** c.-à-d. des fonctions) qui définit son **comportement**.

Un objet est créé à partir d'un **modèle** appelé **classe**. Chaque objet créé à partir de cette classe est une **instance** de la classe en question.

Un objet posséde une **identité** qui permet de distinguer un objet d'un autre objet (son nom, une adresse mémoire).

Notion de classe

Une **classe déclare des propriétés communes** à un ensemble d'objets.

Une classe représentera donc une **catégorie d'objets**.

Elle apparaît comme un **type** ou un *moule* à partir duquel il sera possible de créer des objets.

Notion de visibilité

Le C++ permet de préciser le **type d'accès des membres** (**attributs** et **méthodes**) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.

Il existe aussi un accès **protégé** (`protected`) en C++ que l'on abordera plus tard.

Notion d'encapsulation

L'encapsulation est l'idée de **protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler**.

En respectant ce principe, toutes les variables (attributs) d'une classe seront donc privées.

L'objet est ainsi vu de l'extérieur comme une "boîte noire" possédant certaines propriétés et ayant un comportement spécifié.

C'est le comportement d'un objet qui modifiera son état.

Exemple détaillé n°1 : une classe Point

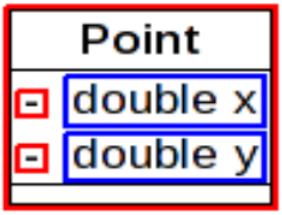
Nous allons successivement découvrir les notions suivantes :
- la modélisation et la déclaration d'une classe
- la construction d'objets d'une classe et la définition de ses fonctions membres
- les services rendus par une classe
- l'accès contrôlé aux membres d'une classe

Modélisation d'une classe

On veut manipuler des **points**. Un **point** est défini par **son abscisse** (`x`) et **son ordonnée** (`y`). L'abscisse et l'ordonnée d'un point sont des **réels** (`double`).

On en sait suffisamment pour modéliser une **classe `Point`** pour nos besoins :

UML	C++

	`class Point`
	`{`
	`  private:`
	`  double x;`
	`  double y;`
	`};`

Le code C++ ci-dessus correspond à la déclaration de la classe `Point`. Elle se placera donc dans un fichier en-tête (*header*) `Point.h`.

Construction d'objets

Pour **créer des objets à partir de cette classe**, il faudra ... **un constructeur** :
- Un constructeur est chargé d'**initialiser un objet de la classe**.
- Il est appelé **automatiquement au moment de la création** de l'objet.
- Un constructeur est une **méthode qui porte toujours le même nom que la classe**.
- Il existe quelques contraintes :
  - Il peut avoir des paramètres, et des valeurs par défaut.
  - Il peut y avoir plusieurs constructeurs pour une même classe.
  - Il n'a jamais de type de retour.

On le **déclare** de la manière suivante :

```
class Point
{
  private:
    double x, y; // nous sommes des attributs de la classe Point

  public:
    Point(double x, double y); // je suis le constructeur de la classe Point
};
```

Point.h

Il faut maintenant **définir** ce constructeur afin qu'il **initialise tous les attributs de l'objet au moment de sa création** :

```
// Je suis le constructeur de la classe Point
Point::Point(double x, double y)
{
   // je dois initialiser TOUS les attributs de la classe
   this->x = x; // on affecte l'argument x à l'attribut x
   this->y = y; // on affecte l'argument y à l'attribut y
}
```

Point.cpp

Le code C++ ci-dessus correspond à la définition du constructeur la classe `Point`. Elle se placera donc dans un fichier C++ `Point.cpp`. On doit faire précéder chaque méthode de `Point::` pour préciser au compilateur que ce sont des membres de la classe `Point`. Le mot clé "`this`" permet de désigner l'adresse de l'objet sur laquelle la fonction membre a été appelée.

On pourra alors **créer nos propres points** :

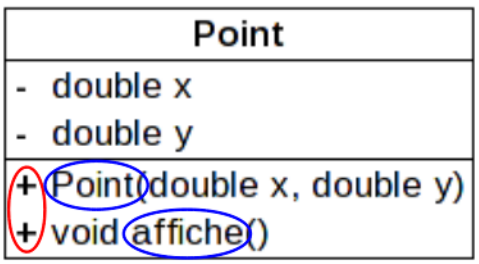
UML	C++
p1 Point x = 1 y = 2	`Point p1(1, 2);`
p2:Point x = 4 y = 0.	`Point p2(4, 0.);`

Les objets `p1` et `p1` sont des instances de la classe `Point`. Un objet possède sa propre existence et un état qui lui est spécifique (c.-à-d. les valeurs de ses attributs).

Les services rendus par une classe

Un point pourra s'**afficher**. On aura donc une **méthode** `affiche()` qui produira un affichage de ce type : `<x,y>`

UML



C++

```
class Point
{
   private:
      double x, y;

   public:
      Point(double x, double y);

      void affiche();
};

// Je suis le constructeur de la
classe Point
Point::Point(double x, double y)
{
   this->x = x;
   this->y = y;
}

// Je suis la méthode publique
affiche()
void Point::affiche()
{
   cout << "<" << x << "," << y <<
">" << endl;
}
```

A l'intérieur de la méthode `affiche()` de la classe `Point`, on peut accéder directement à l'abscisse du point en utilisant la donnée membre `x`. De la même manière, on pourra accéder directement à l'ordonnée du point en utilisant la donnée membre `y`.

On utilisera cette méthode dès que l'on voudra **afficher** les coordonnées d'un point :

```
cout << "P1 = ";
p1.affiche();

cout << "P2 = ";
p2.affiche();
```

Ce qui donnera à l'exécution :

```
P1 = <1,2>
P2 = <4,0>
```

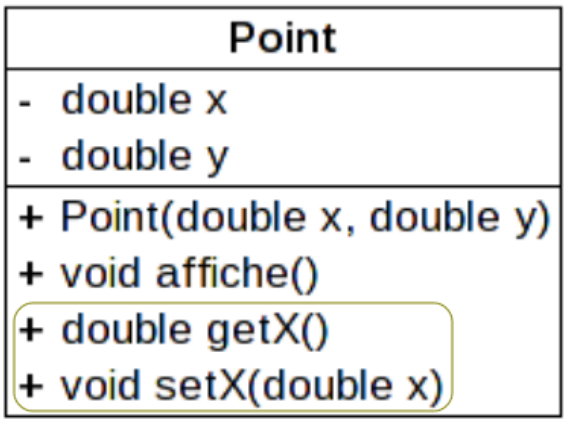
Une méthode publique est un service rendu à l'utilisateur de l'objet.

Les accès contrôlés aux membres d'une classe

Toutes les variables de la classe `Point` étant **privées par respect du principe d'encapsulation**, on veut néanmoins pouvoir connaître son abscisse et pouvoir modifier cette dernière.

Il faut donc créer deux **méthodes publiques** pour **accéder** à l'**attribut** `x` :

UML



C++

```cpp
class Point
{
    private:
        double x, y;

    public:
        Point(double x, double y);
        void affiche();
        double getX() const;
        void setX(double x);

};

// L'accesseur get du membre x
double Point::getX() const
{
    return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
    this->x = x;
}
```

La méthode `getX()` est déclarée constante (`const`). Une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet. Il est conseillé de qualifier `const` toute fonction qui peut l'être car cela garantit qu'on ne pourra appliquer des méthodes constantes que sur un objet constant.

La méthode publique `getX()` est un accesseur (*get*) et `setX()` est un manipulateur (*set*) de l'attribut `x`. L'utilisateur de cet objet ne pourra pas lire ou modifier directement une propriété sans passer par un accesseur ou un manipulateur.

On utilisera ces méthodes pour accéder en lecture ou écriture aux membres privés d'un point :

```
// on peut modifier le membre x de P1
p1.setX(5);
// on peut accéder au membre x de P1
cout << "L'abscisse de P1 est " << p1.getX() << endl;

// on peut accéder au membre x de P2
cout << "L'abscisse de P2 est " << p2.getX() << endl;
```

Ce qui donnera à l'exécution :

```
L'abscisse de P1 est 5
L'abscisse de P2 est 4
```

Constructeur par défaut

Si vous essayer de créer un objet sans lui fournir une abscisse `x` et une ordonnée `y`, vous obtiendrez le message d'erreur suivant :

```
erreur: no matching function for call to Point::Point()'
```

Ce type de constructeur se nomme un **constructeur par défaut**. Son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable.

Le constructeur par défaut de la classe `Point` sera :

```
Point::Point() // Sans aucun paramètre !
{
   x = 0.;
   y = 0.;
}
```

Allocation dynamique d'objet

Pour allouer dynamiquement un objet en C++, on utilisera l'opérateur `new`. Celui-ci renvoyant une adresse où est créé l'objet en question, il faudra un pointeur pour la conserver. Manipuler ce pointeur, reviendra à manipuler l'objet alloué dynamiquement.

Pour libérer la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

```
Point *p3; // je suis pointeur sur un objet de type Point
p3 = new Point(2,2); // j'alloue dynamiquement un objet de type Point
cout << "p3 = ";
p3->affiche(); // Comme pointC est une adresse, je dois utiliser l'opérateur -> pour accéder
     aux membres de cet objet
//p3->setY(0); // je modifie la valeur de l'attribut y de p3
cout << "p3 = ";
(*p3).affiche(); // cette écriture est possible : je pointe l'objet puis j'appelle sa
   méthode affiche()
delete p3; // ne pas oublier de libérer la mémoire allouée pour cet objet
```


Un tableau d'objets

Il est possible de conserver et de manipuler des objets `Point` dans un tableau.

```
// typiquement : les cases d'un tableau de Point
Point tableauDe10Points[10]; // le constructeur par défaut est appelé 10 fois (pour chaque
    objet Point du tableau) !
int i;

cout << "Un tableau de 10 Point : " << endl;
for(i = 0; i < 10; i++)
{
   cout << "P" << i << " = "; tableauDe10Points[i].affiche();
}
cout << endl;
```

Un objet Point constant

Les règles suivantes s'appliquent aux objets constants :
– On déclare un objet constant avec le modificateur `const`
– On ne peut appliquer que des méthodes constantes sur un objet constant
– Un objet passé en paramètre sous forme de référence constante est considéré comme constant

```
const Point p5(7, 8);

cout << "p5 = ";
p5.affiche(); // la méthode affiche() doit être déclarée const [1]
cout << "L'abscisse de p5 est " << p5.getX() << endl;
cout << "L'ordonnée de p5 est " << p5.getY() << endl;
p5.setX(5); // vous ne pouvez pas appeler cette méthode car elle n'est pas const [2]
```

On obtient deux erreurs à la compilation qu'il faudra corriger ([1] et [2]) :

```
erreur: passing 'const Point' as 'this' argument of 'void Point::affiche()' discards
    qualifiers
erreur: passing 'const Point' as 'this' argument of 'void Point::setX(double)' discards
    qualifiers
```

Rendre des services

On doit pouvoir calculer la **distance** entre 2 points et le **milieu** de 2 points.

```
Point p1(7, 8);
Point p2;
double distance = p1.distance(p2);

cout << "p1 = ";
p5.affiche(); // p1 = <7,8>
cout << "p2 = ";
p2.affiche(); // p2 = <0,0>
cout << "La distance entre p1 et p2 est de " << distance << endl; // La distance entre p1 et
     p2 est de 11

Point pointMilieu = p1.milieu(p2);
```



```
cout << "Le point milieu entre p1 et p2 est ";
pointMilieu.affiche(); // Le point milieu entre p1 et p2 est <3.5,4>
```

L'objet `Point` passé en **argument** des méthodes `distance()` et `milieu()` le sera :
- en **référence** ce qui assure de bonnes performances
- et la référence sera **constante**, ce qui garantit que seules des méthodes constantes (ne pouvant pas modifier les attributs) seront appelables sur l'objet passé en argument

Exemple détaillé n°2 : une classe PileChar

Nous allons successivement découvrir les notions suivantes :
– l'utilisation de paramètres par défaut et de liste d'initialisation
– l'écriture d'un destructeur
– l'implémentation d'un constructeur de copie et de l'opérateur d'affectation
– la surcharge d'opérateurs

Une **pile (« stack » en anglais)** est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (*Last In, First Out*)**, ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Une **pile** est utilisée en général pour **gérer un historique de données** (pages webs visitées, ...) ou **d'actions** (les fonctions « Annuler » de beaucoup de logiciels par exemple). La pile est utilisée aussi pour tous les paramètres d'appels et les variables locales des fonctions dans les langages compilés.

Voici quelques fonctions communément utilisées pour manipuler des **piles** :
– « Empiler » : ajoute ou dépose un élément sur la pile
– « Dépiler » : enlève un élément de la pile et le renvoie
– « La pile est-elle vide ? » : renvoie « vrai » si la pile est vide, « faux » sinon
– « La pile est-elle pleine ? » : renvoie « vrai » si la pile est pleine, « faux » sinon
– « Nombre d'éléments dans la pile » : renvoie le nombre d'éléments présents dans la pile
– « Taille de la pile » : renvoie le nombre maximum d'éléments pouvant être déposés dans la pile
– « Quel est l'élément de tête ? » : renvoie l'élément de tête (le sommet) sans le dépiler

Modélisation de la classe PileChar

Le but est de réaliser une **structure de pile pouvant traiter des caractères** (type `char`).

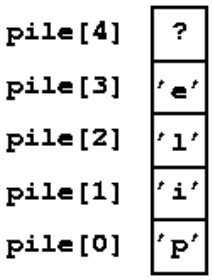
La classe `PileChar` contient **trois données membres privées** :
– deux entiers strictement positifs, nommés `max` et `sommet`, et
– un pointeur sur un caractère, nommé `pile`.

La donnée membre `max` contient la taille de la pile créée pour cette instance de la classe, autrement dit le nombre maximum de caractères qu'il sera possible d'y mettre.

La donnée membre `sommet` indique le numéro de la case dans laquelle on pourra empiler le prochain caractère. Ce n'est donc pas exactement le sommet de la pile, mais un cran au dessus.

Le pointeur sur un caractère `pile` désigne le tableau de caractères, alloué dynamiquement (avec `new`) pour mémoriser le contenu de la pile.

Dans cet exemple, on a successivement empilé les quatre lettres du mot "pile". Ici, `max` vaut 5 (le nombre maximal de lettre empilable). et `sommet` vaut 4 puisque le dernier élément empilé est le 'e' du mot "pile", et que le prochain empilage se fera en `pile[4]`.



Paramètre par défaut

Il y a évidemment un **constructeur** qui prend en paramètre un entier strictement positif pour préciser la taille désirée pour la pile, future valeur pour `max`. On précisera pour ce paramètre, une valeur par défaut de `50` (c'est une **constante** !).

Le langage C++ offre la possibilité d'avoir des **valeurs par défaut pour les paramètres d'une fonction (ou d'une méthode)**, qui peuvent alors être sous-entendus au moment de l'appel.

Cette possibilité, permet d'écrire qu'un seul constructeur profitant du mécanisme de valeur par défaut :

```
#define TAILLE_PAR_DEFAUT 50

// Active les affichages de debuggage pour les constructeurs et destructeur
#define DEBUG

class PileChar
{
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;
   public:
      PileChar(int taille=TAILLE_PAR_DEFAUT); // je suis le constructeur (par défaut) de la
          classe PileChar
};
```

PileChar.h

Liste d'initialisation

Un meilleur moyen d'affecter des valeurs aux données membres de la classe lors de la construction est la **liste d'initialisation**. On va utiliser cette technique pour définir le constructeur de la classe `PileChar` :

```
PileChar::PileChar(int taille/*=TAILLE_PAR_DEFAUT*/) : max(taille), sommet(0) // c'est la
    liste d'initialisation
{
   pile = new char[max]; // allocation dynamique du tableau de caractère
   #ifdef DEBUG
   cout << "PileChar(" << taille << ") : " << this << "\n";
   #endif
}
```

PileChar.cpp

La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup. La liste d'initialisation doit être utilisée pour certains objets qui ne peuvent pas être contruits par défaut : c'est le cas des références et des objets constants.

Destructeur

Le **destructeur** est la **méthode membre appelée automatiquement lorsqu'une instance (objet) de classe cesse d'exister en mémoire** :

- Son rôle est de **libérer toutes les ressources qui ont été acquises lors de la construction** (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"**.
- Il existe quelques contraintes :
  - Il ne possède aucun paramètre.
  - Il n'y en a qu'un et un seul.
  - Il n'a jamais de type de retour.

La forme habituelle d'un destructeur est la suivante :

```
class T {
   public:
      ~T(); // destructeur
};
```

Pour éviter les fuites de mémoire, le destructeur de la classe `PileChar` doit libérer la mémoire allouée au tableau de caractères `pile` :

On le **définit** de la manière suivante :

```
PileChar::~PileChar()
{
   delete [] pile;
   #ifdef DEBUG
   cout << "~PileChar() : " << this << "\n";
   #endif
}
```

PileChar.cpp

Constructeur de copie

Le **constructeur de copie** est appelé dans :
– la création d'un objet à partir d'un autre objet pris comme modèle
– le passage en paramètre d'un objet par valeur à une fonction ou une méthode
– le retour d'une fonction ou une méthode renvoyant un objet

Remarque : toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation (=).

La forme habituelle d'un constructeur de copie est la suivante :

```
class T
{
   public:
      T(const T&);
};
```

Donc pour la classe `PileChar` :

```
PileChar::PileChar(const PileChar &p) : max(p.max), sommet(p.sommet)
{
   pile = new char[max]; // on alloue dynamiquement le tableau de caractère

   unsigned int i;

   // on recopie les éléments de la pile
   for (i = 0; i < sommet ; i++) pile[i] = p.pile[i];
   #ifdef DEBUG
   cout << "PileChar(const PileChar &p) : " << this << "\n";
   #endif
}
```

Deux situations où le constructeur de copie est nécessaire :

```
PileChar pile3a(pile2); // Appel du constructeur de copie pour instancier pile3a

PileChar pile3b = pile1b; // Appel du constructeur de copie pour instancier pile3b
```

Opérateur d'affectation

L'opérateur d'affectation (=) est un opérateur de copie d'un objet vers un autre. L'objet affecté est déjà créé sinon c'est le constructeur de copie qui sera appelé.

La forme habituelle d'opérateur d'affectation est la suivante :

```
class T
{
   public:
      T& operator=(const T&);
};
```


Cet opérateur renvoie une référence sur `T` afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite a=b=c est évaluée comme a=(b=c). Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

La définition de l'opérateur = est la suivante :

```
PileChar& PileChar::operator = (const PileChar &p)
{
   // vérifions si on ne s'auto-copie pas !
   if (this != &p)
   {
      delete [] pile; // on libère l'ancienne pile
      max = p.max;
      sommet = p.sommet;
      pile = new char[max]; // on alloue une nouvelle pile
      unsigned int i;
      for (i = 0; i < sommet ; i++) pile[i] = p.pile[i]; // on recopie les éléments de la
          pile
   }
   #ifdef DEBUG
   cout << "operator= (const PileChar &p) : " << this << "\n";
   #endif
   return *this;
}
```

Ce qui permettra d'écrire :

```
PileChar pile4; // Appel du constructeur par défaut pour créer pile4

pile4 = pile3a; // Appel de l'opérateur d'affectation pour copier pile3a dans pile4
```

Surcharge d'opérateurs

La **surcharge d'opérateur** permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Parmi les nombreux exemples que l'on pourrait citer :
- `myString + yourString` pourrait servir à concaténer deux objets `string`
- `maDate++` pourrait servir à incrémenter un objet `Date`
- `a * b` pourrait servir à multiplier deux objets `Nombre`
- `e[i]` pourrait donner accès à un élément contenu dans un objet `Ensemble`

Les opérateurs C++ que l'on surcharge habituellement :
- Affectation, affectation avec opération (`=`, `+=`, `*=`, etc.) : **Méthode**
- Opérateur « fonction » `()` : **Méthode**
- Opérateur « indirection » `*` : **Méthode**
- Opérateur « crochets » `[]` : **Méthode**
- Incrémentation `++`, décrémentation `--` : **Méthode**
- Opérateur « flèche » et « flèche appel » `->` et `->*` : **Méthode**
- Opérateurs de décalage `<<` et `>>` : **Méthode**
- Opérateurs `new` et `delete` : **Méthode**
- Opérateurs de lecture et écriture sur flux `<<` et `>>` : **Fonction**
- Opérateurs dyadiques genre « arithmétique » (`+`,`-`,`/` etc) : **Fonction**

Les autres opérateurs ne peuvent pas soit être surchargés soit il est déconseillé de le faire.

La **première technique** pour surcharger les opérateurs consiste à les considérer comme des **méthodes** normales de la classe sur laquelle ils s'appliquent.

Le principe est le suivant :

```
A Op B se traduit par A.operatorOp(B)

t1 == t2; // équivalent à : t1.operator==(t2)
t1 += t2; // équivalent à : t1.operator+=(t2)
```

On surcharge les opérateurs `==`, `!=` et `+=` pour la classe `PileChar` :

```
class PileChar
{
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;

   public:
      ...
      bool operator == (const PileChar &p); // teste si deux piles sont identiques
      bool operator != (const PileChar &p); // teste si deux piles sont différentes
      PileChar& operator += (const PileChar &p); // empile une pile sur une autre
};

bool PileChar::operator == (const PileChar &p)
{
   if(max != p.max) return false;
   if(sommet != p.sommet) return false;
   unsigned int i;
   for (i = 0; i < sommet ; i++) if(pile[i] != p.pile[i]) return false;
   return true;
}

bool PileChar::operator != (const PileChar &p)
{
   // TODO
}
```

```
PileChar& PileChar::operator += (const PileChar &p)
{
   unsigned int i, j;

   // Reste-il assez de place pour empiler les caractères ?
   if((sommet + p.sommet) <= max)
   {
      for (i = sommet, j = 0; j < p.sommet ; i++, j++) pile[i] = p.pile[j]; // on recopie
          les éléments de la pile
      sommet += j; // on met à jour le nouveau sommet
   }
   else cerr << "Pile pleine !\n";

   return *this;
}
```

On pourrait aussi surcharger l'opérateur `+=` pour empiler un simple caractère. De la même manière, on surchargerait l'opérateur `-=` pour supprimer `n` caractères depuis le haut de la pile :

```
PileChar pile6;

pile6 += 'a'; // empile le caractère 'a'
pile6 += 'z'; // empile le caractère 'z'
pile6 += 'e'; // empile le caractère 'e'
...
pile6 -= 2; // supprime 2 caractères du haut de la pile (donc 'e' puis 'z')
```

La **deuxième technique** utilise la surcharge d'opérateurs externes sous forme de **fonctions**. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres.

Le principe est le suivant :

```
A Op B se traduit par operatorOp(A, B)

t1 + t2; // équivalent à : operator+(t1, t2)
```

Les opérateurs externes doivent être déclarés comme étant des **fonctions amies** (`friend`) de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

Exemple pour une classe `T` :

```
class T
{
   private:
      int x;

   public:
      friend T operator+(const T &a, const T &b);
};

T operator+(const T &a, const T &b)
{
   // solution n° 1 :
```

```
   T result = a;
   result.x += b.x;
   return result;

   // solution n° 2 : si l'opérateur += a été surchargé
   T result = a;
   return result += b;
}
```

```
T t1, t2, t3; // Des objets de type T

t3 = t1 + t2; // Appel de l'opérateur + puis de l'opérateur de copie (=)

t3 = t2 + t1; // idem car l'opérateur est symétrique
```

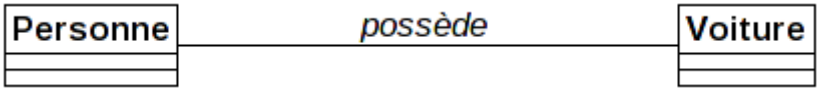
L'avantage de cette syntaxe est que l'opérateur est réellement symétrique, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe.

Notion de relations

Étant donné qu'en POO les objets logiciels interagissent entre eux, il y a donc des **relations** entre les classes.

On distingue trois différents types de **relations** entre les classes :
– l'**association** (trait plein avec ou sans flèche)
– la **dépendance** (flèche pointillée)
– la relation de **généralisation** ou d'**héritage** (flèche fermée vide)

Une association représente une relation sémantique durable entre deux classes. *Exemple :* Une personne peut posséder des voitures. La relation possède est une association entre les classes `Personne` et `Voiture`.



Il existe aussi deux cas particuliers d'**association** que nous allons découvrir :
– l'**agrégation** (trait plein avec ou sans flèche et un losange vide)
– la **composition** (trait plein avec ou sans flèche et un losange plein)

Exemple détaillé n°3 : commandes d'articles

Nous allons successivement découvrir les notions suivantes :
– les relations d'association, d'agrégation et de composition entre classes
– les conteneurs

Cet exemple présente les éléments (briques) logiciels permettant de **traiter des commandes d'articles** (des livres par exemple).

Ces objets logiciels permettront d'**éditer une commande** comme celle-ci :

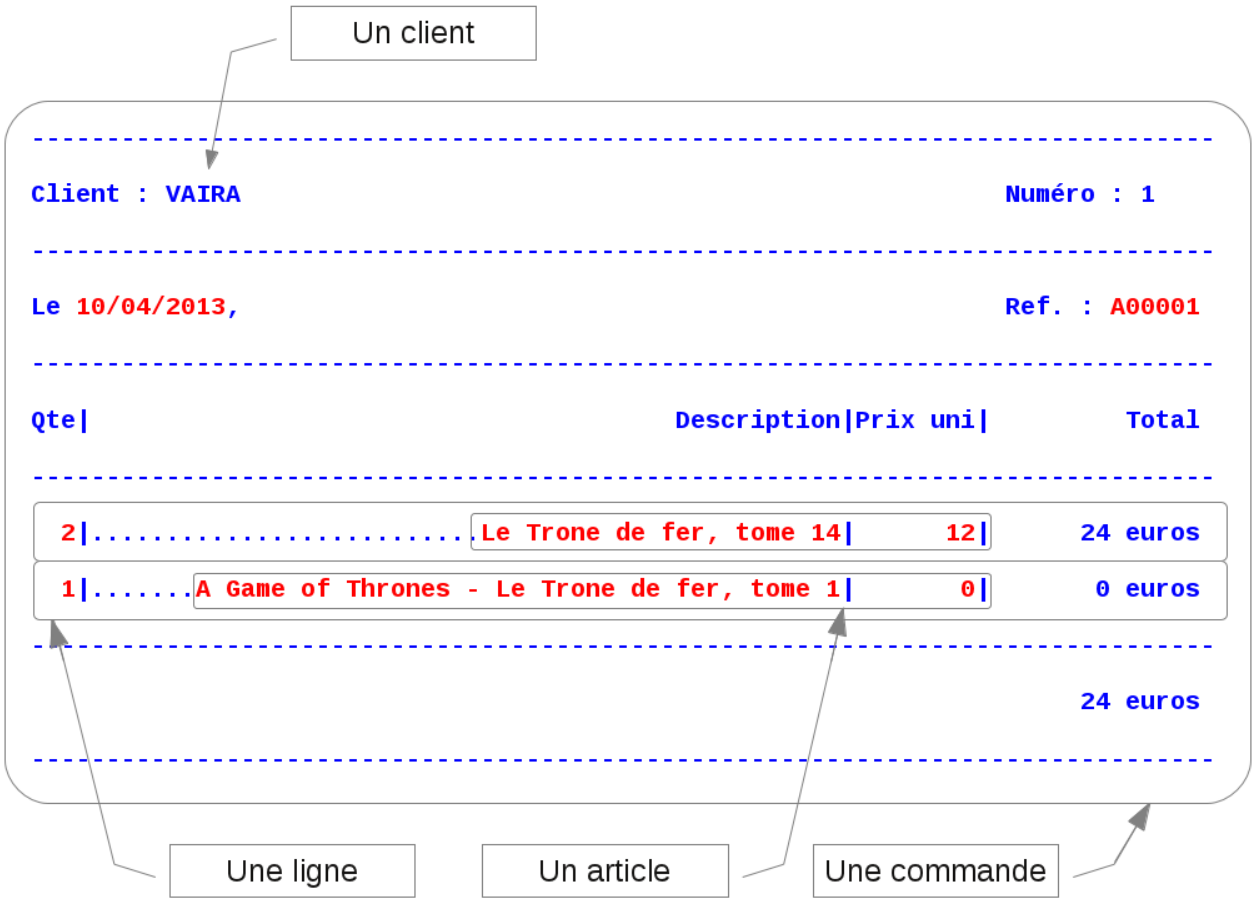
```
--------------------------------------------------------------------------------
Client : VAIRA                                                    Numéro : 1
--------------------------------------------------------------------------------
Le 10/04/2013,                                                    Ref. : A00001
--------------------------------------------------------------------------------
Qte|                                        Description|Prix uni|         Total
--------------------------------------------------------------------------------
  2|.........................Le Trone de fer, tome 14|      12|      24 euros
  1|.......A Game of Thrones - Le Trone de fer, tome 1|       0|       0 euros
--------------------------------------------------------------------------------
                                                                      24 euros
--------------------------------------------------------------------------------
```

Présentation des classes

On a besoin de modéliser quatre classes :

- une classe `Client` qui caractérise une personne qui passe une commande. Un **client** est caractérisé par **son nom** (une chaîne de caractères de type `string`) et **son numéro de client** (un entier de type `int`).
- une classe `Article` décrivant les articles que l'on peut commander. Un **article** est caractérisé par **son titre** (une chaîne de caractères de type `string`) et **son prix** (un réel de type `double`).
- une classe `Commande` qui contient l'ensemble des articles commandés. Une **commande** est caractérisée par **sa référence** (une chaîne de caractères de type `string`) et **sa date** (une chaîne de caractères de type `string`).
- une classe `Ligne` qui correspond à un élément d'un type d'article appartenant à une commande. Une **ligne** d'une commande est caractérisée par **son article** (un objet de type `Article`) et **sa quantité** (un entier de type `long`).

On décompose la commande désirée pour faire apparaître ses composants :

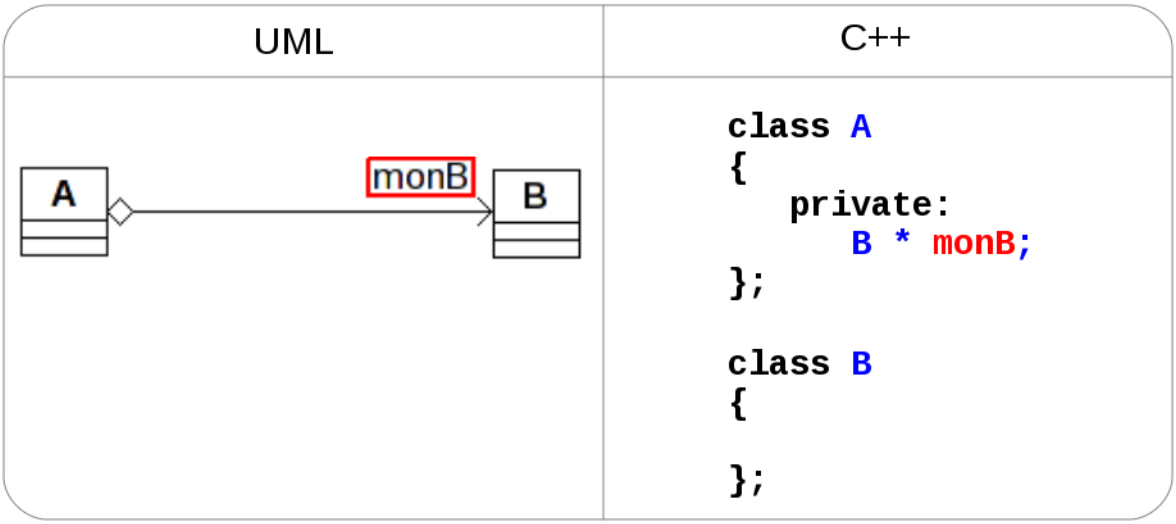


On constate qu'il existe trois relations entre ces classes :

- une association entre les classes `Client` et `Commande`
- une association entre les classes `Ligne` et `Article`
- une association entre les classes `Commande` et `Ligne`

L'agrégation

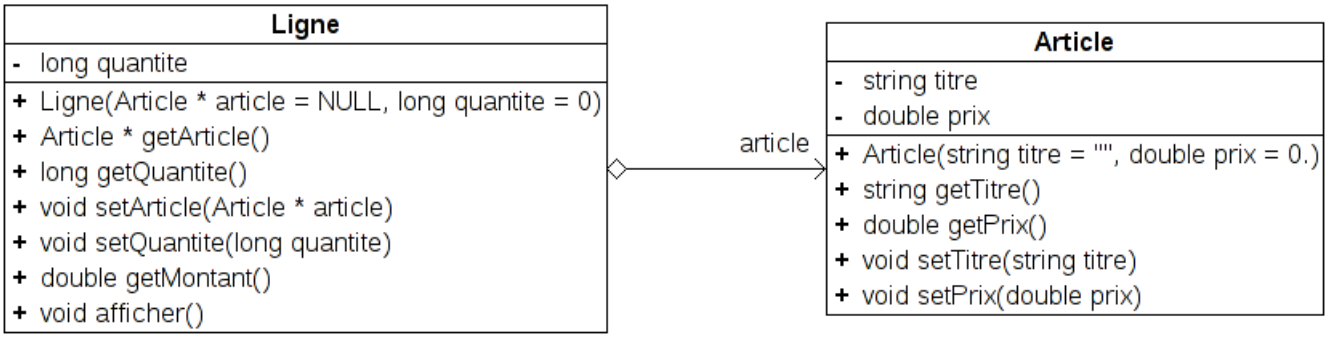
Une **agrégation** est un cas particulier d'association non symétrique exprimant **une relation de contenance**. Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « **contient** » ou « **est composé de** ».



À l'extrémité d'une association, agrégation ou composition, on donne un **nom** : c'est le **rôle** de la relation. Par extension, c'est la manière dont les instances d'une classe voient les instances d'une autre classe au travers de la relation. Ici l'agrégation est nommée `monB` qui est un **attribut** de la classe `A`.

La flèche sur la relation précise la navigabilité. Ici, A « **connaît** » B mais pas l'inverse. Les relations peuvent être bidirectionnel (pas de flèche) ou unidirectionnel (avec une flèche qui précise le sens).

Le diagramme de classe ci-dessous illustre la relation d'agrégation entre la classe `Ligne` et la classe `Article` :



On va tout d'abord écrire la classe `Article` :

```
#ifndef ARTICLE_H
#define ARTICLE_H

class Article
{
   private:
      string titre;
      double prix;
```

```
   public:
      Article(string titre="", double prix=0.); //constructeur
      //Accesseurs
      string getTitre() const;
      double getPrix() const;
      void setTitre(string titre);
      void setPrix(double prix);
};

#endif //ARTICLE_H
```

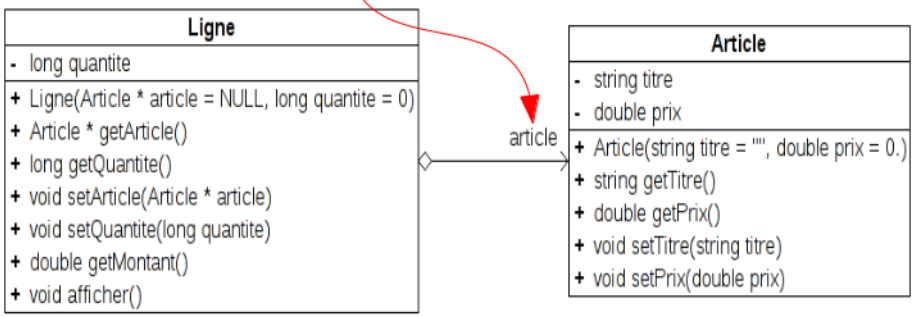
Article.h

Une relation d'agrégation s'implémente généralement par un **pointeur** (pour une relation 1 vers 1) :

C++

```
class Ligne
{
   public:
      // ...
      void setArticle(Article *article); // met
en œuvre l'agrégation

   private:
      // ...
      Article *article; // l'agrégation
};
```

UML



Les accesseurs `getArticle()` et `setArticle()` permettent de gérer la relation `article`. Ici, il est aussi possible d'initialiser la relation au moment de l'instanciation de l'objet de type `Ligne` (cf. son constructeur).

La déclaration de la classe `Ligne` intégrant la relation d'agrégation sera :

```
#ifndef LIGNE_H
#define LIGNE_H

class Article; // je ''déclare'' : Article est une classe ! (1)

class Ligne
{
   private:
      Article *article;
```

```
      long quantite;

   public:
      Ligne(Article *article=NULL, long quantite=0);
      Article * getArticle() const;
      long getQuantite() const;
      void setArticle(Article *article);
      void setQuantite(long quantite);
      double getMontant() const;
      void afficher();
};

#endif //LIGNE_H
```

Ligne.h

(1) Cette ligne est obligatoire pour indiquer au compilateur que `Article` est de type `class` et permet d'éviter le message d'erreur suivant à la compilation : `erreur : 'Article' has not been declared`

La définition (incomplète) de la classe `Ligne` est la suivante :

```
#include <iostream>
#include <iomanip>

using namespace std;

#include "Ligne.h"
#include "Article.h" // accès à la déclaration complète de la classe Article (2)

Ligne::Ligne(Article *article/*=NULL*/, long quantite/*=0*/)
{
   this->article = article; // initialise la relation d'agrégation
   this->quantite = quantite;
}

// etc ...
```

Ligne.cpp

(2) Sans cette ligne, on va obtenir des erreurs à la compilation car celui-ci ne connaît pas "suffisamment" le type `Article` : `erreur : invalid use of incomplete type 'struct Article'`. Pour corriger ces erreurs, il suffit d'**inclure la déclaration (complète) de la classe `Article`** qui est contenue dans le **fichier d'en-tête** (*header*) `Article.h`

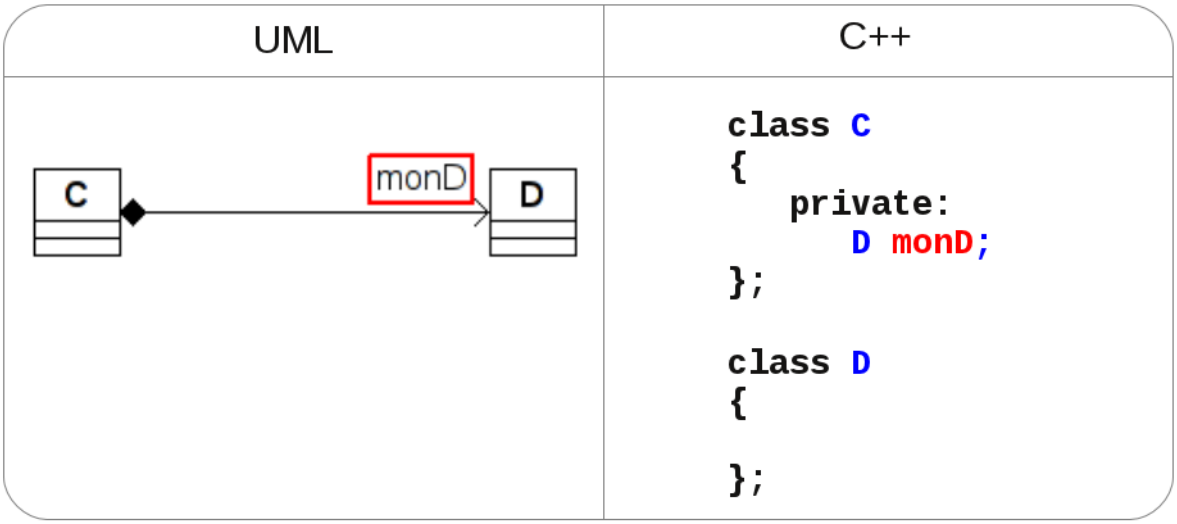
La composition

Une **composition** est une agrégation plus forte signifiant **« est composée d'un »** et impliquant :
- une partie ne peut appartenir qu'à un seul composite (agrégation non partagée)
- la destruction du composite entraîne la destruction de toutes ses parties (il est responsable du cycle de vie de ses parties).

La relation de composition correspond bien à notre besoin car, quand on devra supprimer une commande, on supprimera chaque ligne de celle-ci. D'autre part, une ligne d'une commande ne peut être partagée avec une autre commande : elle est lui est propre.

La relation de composition ne serait pas un bon choix pour la relation entre `Ligne` et `Article`. En effet, lorsqu'on supprime une ligne d'une commande, on ne doit pas supprimer l'article correspondant qui reste commandable par d'autres clients. D'autre part, un même article peut se retrouver dans plusieurs commandes (heureusement pour les ventes !). Donc, l'agrégation choisie précédemment était bien le bon choix.

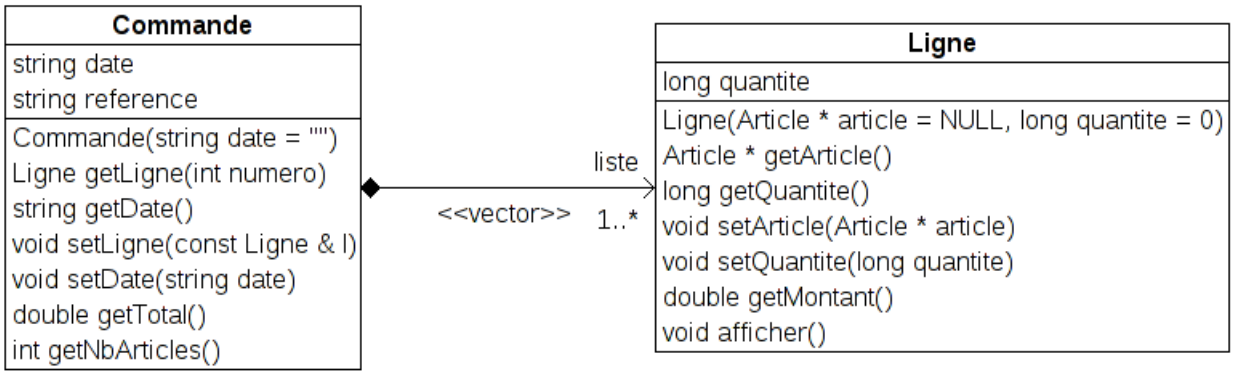
La **composition** se représente de la manière suivante en UML :



Aux extrémités d'une association, agrégation ou composition, il est possible d'y indiquer une **multiplicité** (ou **cardinalité**) : c'est pour préciser le nombre d'instances (objets) qui participent à la relation. Ici la composition est implicitement de 1 vers 1.

Une multiplicité peut s'écrire : n (exactement n, un entier positif), n..m (n à n), n..* (n ou plus) ou * (plusieurs).

Le diagramme de classe ci-dessous illustre la relation de composition entre la classe `Commande` et la classe `Ligne` :



Dans notre cas, une commande peut contenir une (1) ou plusieurs (*) lignes. Pour pouvoir conserver plusieurs lignes, on va utiliser un **conteneur** de type `vector` (indiqué dans le diagramme UML ci-dessus par un stéréotype).

Rappel sur la notion de **vector** (cf. www.cplusplus.com/reference/vector/vector/) : Un *vector* est un **tableau dynamique** où il est particulièrement aisé d'accéder directement aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin. A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.

On n'apporte aucune modification à la classe `Ligne` existante. On va donc maintenant déclarer la classe `Commande` :

```
#ifndef COMMANDE_H
#define COMMANDE_H

#include <vector>

using namespace std;

#include "Ligne.h" // ici il faut un accès à la déclaration complète de la classe Ligne (3)

class Commande
{
   private:
      string reference;
      string date;
      vector<Ligne> liste; // la composition 1..*

   public:
      Commande(string reference="", string date="");
      string getReference() const;
      void setReference(string reference);
      string getDate() const;
      void setDate(string date);
      Ligne getLigne(int numero) const;
      void setLigne(const Ligne &l);
      double getTotal();
      int getNbArticles() const;
```

```
     void afficher();
};

#endif //COMMANDE_H
```

Commande.h

(3) Cette ligne est obligatoire ici car le compilateur a besoin de "connaître complètement" le type `Ligne` car des objets de ce type vont devoir être construits.

La définition (incomplète) de la classe `Commande` est la suivante :

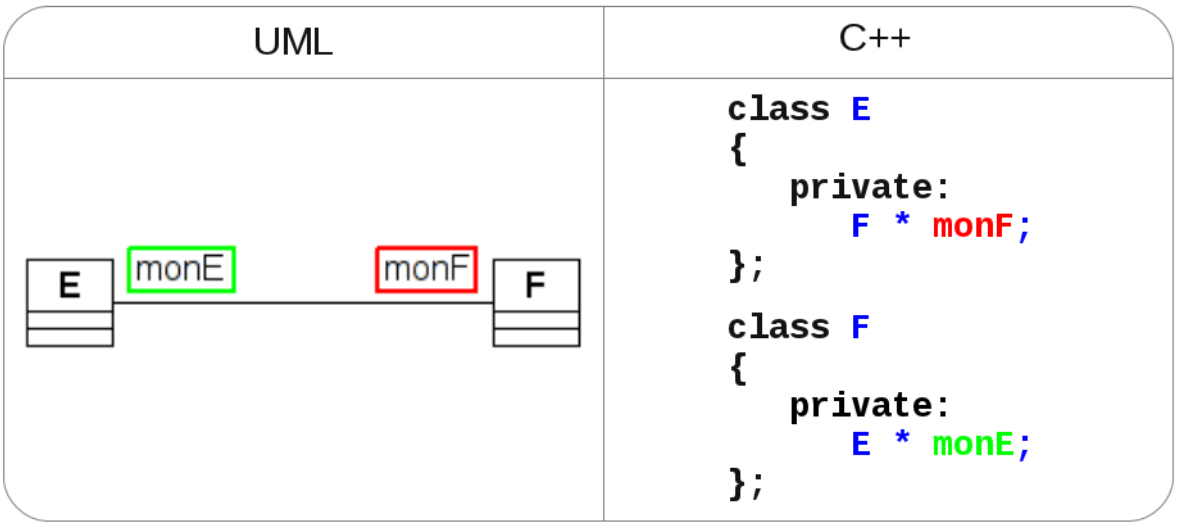
```
#include <iostream>
#include <iomanip>

using namespace std;

#include "Commande.h"

Commande::Commande(string reference/*=""*/, string date/*=*/)
{
}

// etc ...
```

Commande.cpp

L'association

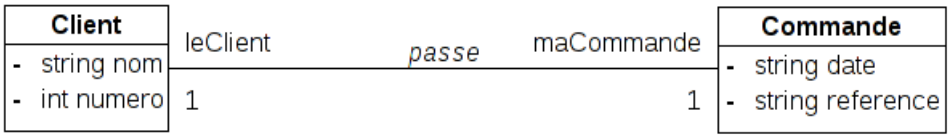
Une **association** représente une **relation sémantique durable entre deux classes**. Les associations peuvent donc être nommées pour donner un sens précis à la relation.

L'**association** se représente de la manière suivante en UML :

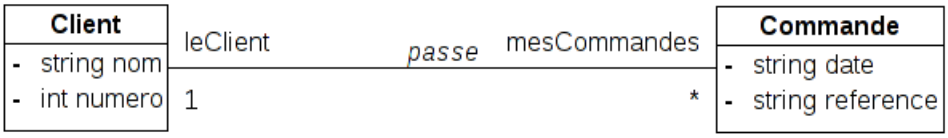


Ici, la relation est bidirectionnelle (pas de flèche), on a une navigabilité dans les deux sens. Ici, A « **connaît** » B et B « **connaît** » A. On peut remarquer que l'association se code de la même manière qu'une agrégation.

Ce diagramme de classe ci-dessous illustre une relation d'association 1 vers 1 entre `Client` et `Commande` :



Ce diagramme de classe ci-dessous illustre une relation d'association 1 vers plusieurs entre `Client` et `Commande` :



Membres statiques

Attributs statiques

Un membre donnée déclaré avec l'attribut `static` est **partagé par tous les objets de la même classe**. Il existe même lorsque aucun objet de cette classe n'a été créé.

Un membre donnée statique doit être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

En général, son initialisation se fait dans le fichier `.cpp` de définition de la classe.

On va utiliser un membre statique `nbPoints` pour **compter le nombre d'objets `Point` créés à un instant donné**.

On **déclare** un membre donnée statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   ...
};
```

Point.h

Il faut maintenant `initialiser` ce membre statique et `compter/décompter` le nombre d'objets créés et détruits :

```
int Point::nbPoints = 0; // initialisation d'un membre statique

Point::Point() // Constructeur
{
   x = 0.;
   y = 0.;
   nbPoints++; // un objet Point de plus
}

...

Point::~Point() // Destructeur
{
   nbPoints--; // un objet Point de moins
}
```

Point.cpp

Tous les constructeurs de la classe `Point` doivent incrémenter le membre statique `nbPoints` !

Fonctions membres statiques

Lorsqu'une fonction membre a **une action indépendante d'un quelconque objet de sa classe**, on peut la déclarer avec l'attribut `static`.

Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Les fonctions membre statiques :
- ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.
- peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

On va utiliser une fonction membre statique `compte()` pour **connaître le nombre d'objets `Point` existants à un instant donné**.

On **déclare** une fonction membre statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   public:
      ...
      static int compte(); // je suis une méthode statique

};
```

Point.h

Il faut maintenant `définir` cette méthode statique :

```
// je retourne le nombre d'objets Point existants à un instant donné
int Point::compte()
{
   return nbPoints;
}
```

Point.cpp

La surcharge des opérateurs de flux << et >>

Rappels : Un **flot** est un **canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information**. Ce canal est associé à un périphérique ou à un fichier.

Un **flot d'entrée** est un objet de type `istream` tandis qu'un **flot de sortie** est un objet de type `ostream`.

Le flot `cout` est un flot de sortie prédéfini connecté à la sortie standard `stdout`. De même, le flot `cin` est un flot d'entrée prédéfini connecté à l'entrée standard `stdin`.

On surchargera les opérateurs de flux << et >> pour une classe quelconque, sous forme de **fonctions amies**, en utilisant ces « canevas » :

```
ostream & operator << (ostream & sortie, const type_classe & objet1)
{
   // Envoi sur le flot sortie des membres de objet en utilisant
   // les possibilités classiques de << pour les types de base
   // c'est-à-dire des instructions de la forme :
   // sortie << ..... ;

   return sortie ;
}

istream & operator >> (istream & entree, type_de_base & objet)
{
   // Lecture des informations correspondant aux différents membres de objet
   // en utilisant les possibilités classiques de >> pour les types de base
   // c'est-à-dire des instructions de la forme :
   // entree >> ..... ;

   return entree ;
}
```

Si on implémente la surcharge de l'opérateurs de flux de sortie << pour qu'il affiche un objet `Point` de la manière suivante : `<x,y>`, on pourra alors écrire :

```
Point p0, p1(4, 0.0), p2(2.5, 2.5);

cout << "P0 = " << p0 << endl;
cout << "P1 = " << p1 << endl;
cout << "P2 = " << p2 << endl;
```

Ce qui donnera :

```
P0 = <0,0>
P1 = <4,0>
P2 = <2.5,2.5>
```

Pour obtenir cela, on écrira :

```
ostream & operator << (ostream & os, const Point & p)
{
   os << "<" << p.x << "," << p.y << ">";
   return os;
}
```

Idem pour la surcharge de l'opérateurs de flux d'entrée `>>` pour qu'il réalise la saisie d'un objet `Point` de la manière suivante : `<x,y>`.

```
cout << "Entrez un point : ";
cin >> p0;

if (! cin)
{
   cout << "ERREUR de lecture !\n";
   return -1;
}

cout << "P0 = " << p0 << endl;
```

Ce qui donnera :

```
Entrez un point : <2,6>
P0 = <2,6>

Entrez un point : <s,k>
ERREUR de lecture !
```


Notion d'héritage

L'**héritage** est un **concept fondamental de la programmation orientée objet**. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.

L'héritage permet **d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise**. Il permet donc **la spécialisation ou la dérivation de types**.

B **hérite** de A : "un B **est** un A avec des choses en plus". Toutes les instances de B sont aussi des instances de A.

On dit aussi que B dérive de A. A est une généralisation de B et B est une spécialisation de A.

Propriétés de l'héritage

L'héritage est **une relation entre classes** qui a les propriétés suivantes :
- si B hérite de A et si C hérite de B alors C hérite de A
- une classe ne peut hériter d'elle-même
- si A hérite de B, B n'hérite pas de A
- il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C
- le C++ permet à une classe C d'hériter des propriétés des classes A et B (héritage multiple)

Notion de visibilité pour l'héritage

*Rappels : Le C++ permet de préciser le **type d'accès des membres** (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets.*

Il faut maintenant tenir compte de la situation d'héritage :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **non par ceux d'une autre classe même dérivée**.
- **protégé** (*protected*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **par ceux d'une classe dérivée**.

Notion de redéfinition

En utilisant l'héritage, il est possible **d'ajouter des caractéristiques**, **d'utiliser les caractéristiques héritées** et de **redéfinir les méthodes héritées**.

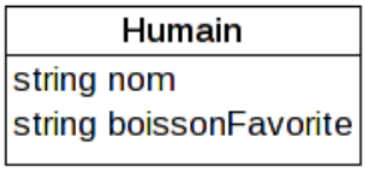
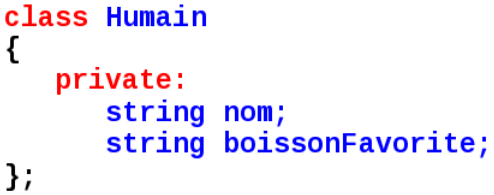
Généralement, cette **redéfinition (*overriding*)** se fait par **surcharge** et permet de modifier le comportement hérité.

Exemple détaillé n°4 : des dames coquettes

On désire réaliser des programmes orientés objet en C++ qui **raconteront des histoires dans lesquelles un cowboy rencontre une dame coquette** :

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky
Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une
jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de coca-cola !
GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```

Les intervenants de nos histoires sont tous des humains. Notre **humain** est caractérisé par son **nom** et sa **boisson favorite**. La boisson favorite d'un humain est, **par défaut**, de l'eau.

UML	C++
	


```
class Humain
{
   private:
      string nom;
      string boissonFavorite;
};
```


Un humain pourra **parler**. On aura donc une **méthode** `parle(texte)` qui affiche :

`(nom de l'humain) -- texte`

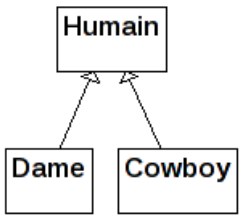
UML	C++

Humain
string nom
string boissonFavorite
Humain(const string nom = "", const string boissonFavorite = "eau")
void parle(const string texte)
	`class Humain`
	`{`
	`  private:`
	`  string nom;`
	`  string boissonFavorite;`
	`  public:`
	`  Humain(const string nom="", const string boissonFavorite="eau");`
	`  void parle(const string texte);`
	`};`
	
	`void Humain::parle(const string texte)`
	`{`
	`  cout << "(" << nom << ") -- " << texte << endl;`
	`}`

Un **humain** pourra également **se présenter** (il dit bonjour, son nom, et indique sa boisson favorite), et **boire** (il dira « Ah ! un bon verre de (sa boisson favorite) ! GLOUPS ! »). On veut aussi pouvoir connaître le **nom** d'un **humain** mais il n'est pas possible de le modifier.

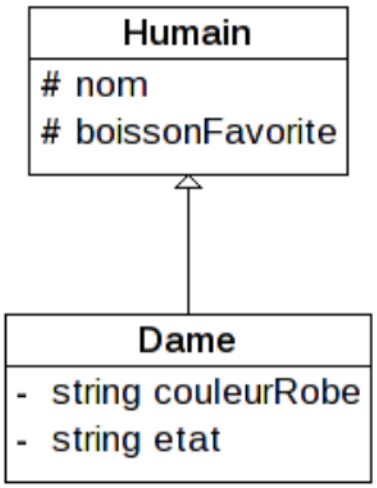
Les **dames** et les **cowboys** sont tous des **humains**. Ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces **deux classes d'humains**.

On va donc réaliser la modélisation suivante en utilisant l'héritage :



Une **dame** est caractérisée par la **couleur de sa robe** (une chaîne de caractères), et par son **état** (libre ou captive).

UML



C++

```
class Humain
{
  public:
      Humain(const string nom="", const
string boissonFavorite="eau");

  protected:
      string nom;
      string boissonFavorite;
};

// La classe Dame hérite de la classe Humain
class Dame : public Humain
{
   public:
      Dame(const string nom="", const string
boissonFavorite="lait", const string
couleurRobe="blanche");

   private:
      string couleurRobe;
      string etat;
};

// Le constructeur Dame appelle le constructeur
Humain puis intialise ses propres attributs
Dame::Dame(const string nom, const string
boissonFavorite, const string couleurRobe) :
Humain(nom, boissonFavorite),
couleurRobe(couleurRobe), etat("libre")
{ }
```

Il est nécessaire d'indiquer une visibilité `protected` aux membres `nom` et `boissonFavorite` pour permettre aux objets de la classe `Dame` d'accéder à ces membres.

Elle peut également **changer de robe** (tout en s'écriant « Regardez ma nouvelle robe (couleur de la robe) ! »). On désire aussi changer le mode de présentation des dames car une dame ne pourra s'empêcher de parler de la couleur de sa robe. Et quand on demande son **nom** a une **dame**, elle devra répondre « Miss (son nom) ».

Il faut donc **redéfinir les méthodes héritées** `getNom()` et `sePresente()` pour obtenir le comportement suivant :

```
Dame jenny("Jenny");

jenny.sePresente();
```

devra donner :

```
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
```

On déclare la classe `Dame` :

```
class Dame : public Humain
{
   public:
      // Constructeurs et Destructeur
      Dame(const string nom="", const string boissonFavorite="lait", const string
          couleurRobe="blanche");

      // Accesseurs
      string getNom() const; // surcharge
      string getEtat() const;

      // Services
      void sePresente() const; // surcharge
      void changeDeRobe(const string couleurRobe);

   private:
      string couleurRobe;
      string etat;
};
```

dame.h

On définit les méthodes de la classe `Dame` :

```
#include "dame.h"

// Constructeur
Dame::Dame(const string nom/*=""*/, const string boissonFavorite/*="lait"*/, const string
    couleurRobe/*="blanche"*/) : Humain(nom, boissonFavorite), couleurRobe(couleurRobe),
    etat("libre")
{
}

// Accesseurs
string Dame::getNom() const
{
   return "Miss " + nom;
}

string Dame::getEtat() const
{
   return etat;
}

// Services
void Dame::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " et j'ai une jolie
       robe " << couleurRobe << endl;
}

void Dame::changeDeRobe(const string couleurRobe)
{
   this->couleurRobe = couleurRobe;
```



```
   cout << "(" << nom << ") -- " << "Regardez ma nouvelle robe " << couleurRobe << " !" <<
      endl;
}
```

dame.cpp

Un **cowboy** est un **humain** qui est caractérisé par sa popularité (0 pour commencer) et un adjectif le caractérisant ("vaillant" par défaut). On désire aussi changer le mode de présentation des cowboys. Un cowboy dira ce que les autres disent de lui (son adjectif).

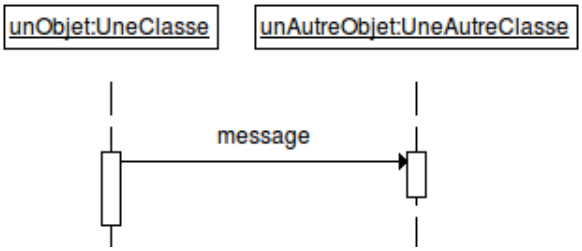
De même, on peut donner une boisson par défaut à chaque sous-classe d'humain : du lait pour les dames et du whisky pour les cowboys ...

Notion de messages

Rappels : La programmation orientée objet consiste à ***définir des objets logiciels et à les faire interagir entre eux****.*

Un objet est une structure de données encapsulées qui répond à un **ensemble de messages**. Cette **structure de données (ses attributs) définit son état** tandis que l'**ensemble des messages (ses méthodes) décrit son comportement**.

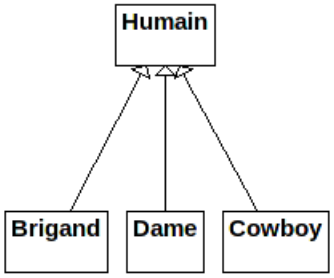
L'ensemble des messages forme ce que l'on appelle l'**interface de l'objet**. Les objets interagissent entre eux en **s'échangeant des messages**.



La réponse à la réception d'un message par un objet est appelée **une méthode**. Une **méthode est donc la mise en oeuvre du message** : elle décrit la réponse qui doit être donnée au message.

Exemple détaillé n°5 : des brigands, des dames et des cowboys

Rappel : Les ***brigands****, les* ***dames*** *et les* ***cowboys*** *sont tous des* ***humains****.*



Une **dame** peut **se faire kidnapper** (auquel cas elle **hurle**), **se faire libérer** par un **cowboy** (elle **remercie** alors le **héros** qui l'a libérée).

La nouvelle classe `Dame` sera alors la suivante :

Dame
- string couleurRobe - string etat
+ Dame(const string nom = "", const string boissonFavorite = "lait", const string couleurRobe = "blanche") + string getNom() + string getEtat() + void sePresente() + void seFaitKidnapper() + void seFaitLiberer(Cowboy & cowboy) + void changeDeRobe(const string couleurRobe) - void hurle() - void remercie(const Cowboy & heros)

Un **brigand** peut **kidnapper** une **dame** (auquel cas, il s'exclame « Ah ah ! (nom de la dame), tu es mienne désormais ! ». Il peut également **se faire emprisonner** par un **cowboy** (il s'écrit alors « Damned, je suis fait ! (nom du cowboy), tu m'as eu ! ». On dispose également d'une fonction pour connaître la récompense obtenue en cas de capture.

La classe `Brigand` sera alors la suivante :

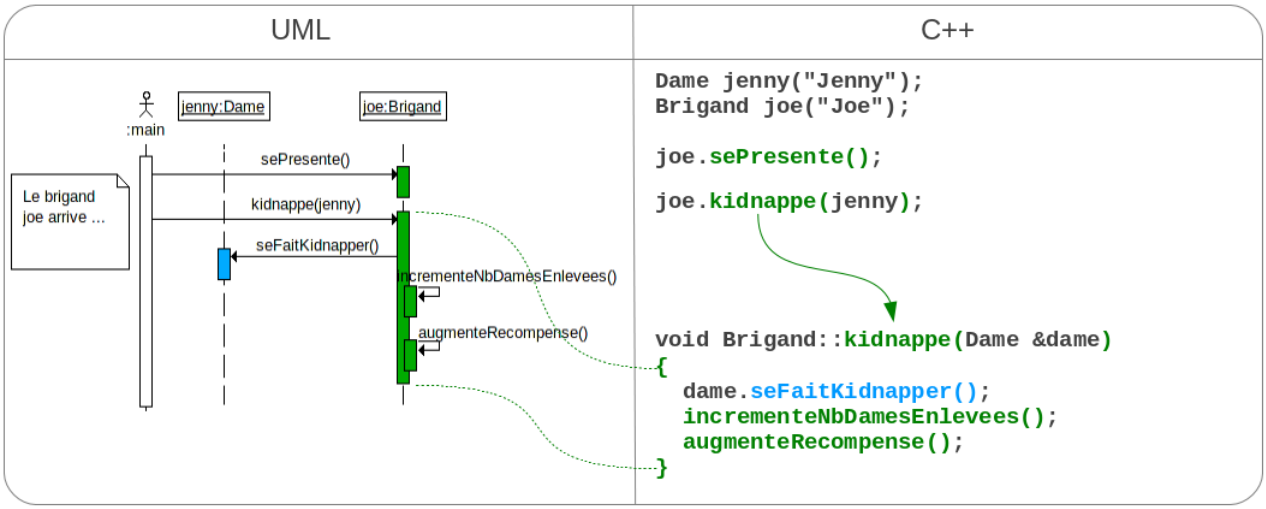
Brigand
- string comportement - int nbDamesEnlevees - int recompense - bool enPrison
+ Brigand(const string nom = "", const string boissonFavorite = "tord-boyaux", const string comportement = "méchant") + string getComportement() + int getNbDamesEnlevees() + int getRecompense() + void sePresente() + void kidnappe(Dame & dame) + void seFaitEmprisonne(Cowboy & cowboy) + void augmenteRecompense(const int prix = 100) + void diminueRecompense(const int prix = 100) + bool estEnPrison()

Un **cowboy** peut **tirer** sur un **brigand**. Un commentaire indique alors « Le (adjectif) (nom) tire sur (nom du méchant). PAN ! » et le cowboy s'exclame « Prend ça, rascal ! ». Il peut également **libérer** une **dame**.

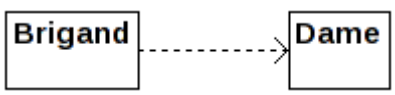
La nouvelle classe `Cowboy` sera alors la suivante :

Cowboy
- int popularite - string qualite
+ Cowboy(const string nom = "", const string boissonFavorite = "whisky", const string qualite = "vaillant") + int getPopularite() + void setPopularite(const int popularite) + string getQualite() + void setQualite(const string qualite) + void sePresente() + void incrementePopularite() + void decrementePopularite() + void tire(const Brigand & brigand) + void emprisonne(Brigand & brigand) + void libere(Dame & dame)

Prenons la situation où un **brigand kidnappe une dame** : les objets `joe` et `jenny` doivent interagir entre eux



Ici l'objet joe de type `Brigand` "utilise" les services de l'objet jenny de type `Dame` par l'intermédiaire de la méthode `seFaitKidnapper()`. Une relation d'utilisation est une dépendance entre classes. La plupart du temps, les dépendances servent à montrer qu'une classe utilise une autre comme argument dans la signature d'une méthode. On parle aussi de lien temporaire car il ne dure que le temps de l'exécution de la méthode. Cela peut être aussi le cas d'un objet local à une méthode.



Une dépendance s'illustre par une flèche en pointillée dans un diagramme de classes en UML.

Les dépendances entre classes

Ces dépendances vont avoir des répercussions sur le code des classes.

Par exemple, lorsqu'on va déclarer la méthode `kidnappe()` dans la classe `Brigand`, il va falloir écrire :

```
class Brigand : public Humain
{
   ...
   void kidnappe(Dame &dame);
   ...
};
```

brigand.h

On va obtenir une erreur à la compilation car le type `Dame` n'a pas été déclaré :

```
erreur: 'Dame' has not been declared
```

Pour corriger cette erreur, il suffit d'indiquer au compilateur que le type **`Dame` est une classe** puisque c'est ce qu'il veut savoir :

```
class Dame; // je ''déclare'' : Dame est une classe !

class Brigand : public Humain
{
   ...
   void kidnappe(Dame &dame);
   ...
};
```

brigand.h

La définition de la méthode `kidnappe()` entraîne de nouveaux problèmes :

```
...
void Brigand::kidnappe(Dame &dame) // <- problème : appel du constructeur Dame ?!
{
   dame.seFaitKidnapper(); // <- problème : appel de seFaitKidnapper() ?!
   nbDamesEnlevees++;
   augmenteRecompense();
   cout << "(" << nom << ") -- " << "Ah ah ! " << dame.getNom() << ", tu es mienne désormais
        !" << endl; // <- problème : appel de getNom() ?!
}
...
```

brigand.cpp

On va obtenir des erreurs à la compilation car celui-ci ne connaît pas "suffisamment" le type `Dame` :

```
erreur: invalid use of incomplete type 'struct Dame'
...
```

Pour corriger ces erreurs, il suffit d'**inclure la déclaration (complète) de la classe `Dame`** qui est contenue dans le **fichier d'en-tête** (*header*) `Dame.h` :

```
#include "brigand.h"
#include "dame.h" // accès à la déclaration complète de la classe Dame
...
void Brigand::kidnappe(Dame &dame)
```



```
{
   dame.seFaitKidnapper();
   nbDamesEnlevees++;
   augmenteRecompense();
   cout << "(" << nom << ") -- " << "Ah ah ! " << dame.getNom() << ", tu es mienne désormais
        !" << endl;
}
...
```

brigand.cpp

Notion de redéfinition

Il ne faut pas mélanger la redéfinition et la surdéfinition :
- Une **surdéfinition (ou surcharge)** permet **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente**.
- Une **redéfinition (*overriding*)** permet **de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer**. Elle doit avoir une signature rigoureusement identique à la méthode parente.

Un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire ou de la compléter.

Notion de polymorphisme

Le **polymorphisme** représente **la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation**.

On voit donc apparaître ici le concept de **polymorphisme** : **choisir en fonction des besoins quelle méthode appeler et ce au cours même de l'exécution**.

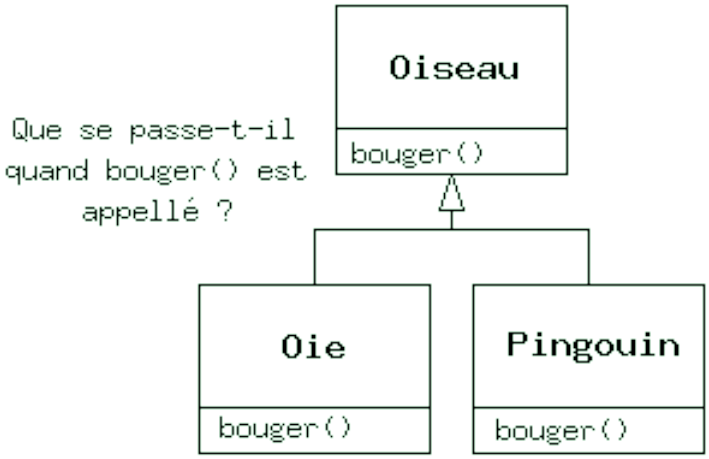
Le **polymorphisme** est implémenté **en C++** avec **les fonctions `virtual` et l'héritage**.

Notion de fonctions virtuelles

Pour créer une fonction membre `virtual`, il suffit de faire précéder la déclaration de la fonction du mot-clef `virtual`. Seule, la déclaration nécessite ce mot-clef, pas la définition. Si une fonction est déclarée `virtual` dans la classe de base, elle est `virtual` dans toutes les classes dérivées.

La redéfinition d'une fonction virtuelle dans une classe dérivée est généralement appelée **redéfinition (*overriding*)**.

Exemple : Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'`Oiseau`, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ? Car la méthode `bouger()` de la classe `Oiseau` a été déclarée `virtual`. Puis les classes `Oie` et `Pingouin` l'ont redéfinie pour obtenir le bon comportement.



On commence à posséder de nombreux types d'humain. On va introduire une fonction qui leur permettra de se présenter :

```
void presentezVous(const Humain &humain)
{
   humain.sePresente();
}
```

presentezVous()

Ce qui permettra à chaque humain de se présenter correctement :

```
Cowboy lucky("Lucky Luke", "coca-cola");
Dame jenny("Jenny");
Brigand joe("Joe");
Barman robert("Robert");

// 1. les présentations des personnages de l'histoire
presentezVous(lucky);
presentezVous(jenny);
presentezVous(joe);
presentezVous(robert);
```

Les présentations

Il n'y a pas d'erreurs à la compilation puisque les **cowboys**, les **dames**, les **barmans** et même les **brigands** sont tous des **humains**.

Par contre, on n'obtient pas ce que l'on désire à l'exécution :

```
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Jenny et j'aime le lait
(Joe) -- Bonjour, je suis Joe et j'aime le tord-boyaux
(Robert) -- Bonjour, je suis Robert et j'aime la bière
```

En effet, c'est la méthode `sePresente()` de la classe `Humain` qui est appelée et non celle des classes `Cowboy`, `Dame`, `Barman` et `Brigand`.

Pour corriger cela, il suffit de déclarer la méthode `sePresente()` comme **virtuelle (`virtual`)** dans la classe `Humain` :


```
class Humain
{
   ...
   virtual void sePresente() const;
   ...
};
```

humain.h

Puis, chaque classe héritée de `Humain` pourra redéfinir la méthode `sePresente()` :

```
class Dame : public Humain
{
   ...
   void sePresente() const;
   ...
};

void Dame::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " et j'ai une jolie
      robe " << couleurRobe << endl;
}
```

dame.h

On obtient maintenant un **comportement polymorphe** : la "bonne méthode" `sePresente()` est appelée en fonction du type de l'objet à l'exécution

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
(Joe) -- Bonjour, je suis Joe le méchant et j'aime le tord-boyaux.
(Robert) -- Bonjour, je suis Robert le barman du Saloon Robert et j'aime la bière
```

