Programmation Orientée Objets : deuxième partie

Table des matières

Table des exemples, figures et illustrations

Remerciement : la première version de ce document avait été réalisée, avec le concour sdes enseignants, par *M. Sadou Salah*, ATER au dépt. MIS dans les années 90. IQu'il soit remercié pour ce travail
Depuis, le dit document a subi plusieurs mises à jours par les enseignants du même département.

Gestion des erreurs : exceptions

Habituellement, en cas d'erreur, on affiche un message d'erreur et on termine le programme (cas d'erreur grave) par éventuellement un code de retour ou alors on décide d'un traitement particulier. Dans cette gestion d'erreur passive, il faut traiter les erreurs possibles de chaque instruction ou d'appel de fonction. De plus, l'utilisateur n'a en général pas la possibilité d'intervenir pour éventuellement supprimer les causes de l'erreur. Les fonctions pourraient avoir un paramètre "compte rendu" spécial (vrai = calcul réussi, faux = problème survenu) et/ou on pourra terminer par *exit* (ou *perror*).

Le langage C++ permet de gérer les erreurs et situations anormales d'une manière plus appropriée. Cette gestion est basée sur les **exceptions**.
Une exception est un signal particulier (nommé et typé) qu'une fonction peut transmettre vers l'extérieur. Ce signal est capté et traité par un traitant d'exception prévu au niveau du bloc courant ou du bloc englobant, le bloc supérieur, etc. Si rien n'est prévu, le programme s'arrête (exception renvoyée au système d'exploitation).

Remarque : les exceptions permettent une prise en charge correcte des fonctions partielles dans les Types de données abstraits (TDA).

Le format général de la gestion des exceptions est le suivant :

try {une séquence ou un appel de fonction qui transmet (lève) éventuellement une ou plusieurs exceptions par **throw** exception}
catch(exception1)
{traiter l'exception1}
.....
catch(exceptionN)
{traiter l'exceptionN}

Chaque traitant (suite de *catch*) possède un argument spécifique qui permet de reconnaître un type particulier d'exception.
Une exception est un objet de type quelconque (type de base ou classe définie par l'utilisateur).
Lorsqu'une exception est levée, le contrôle est transmis au traitant de cette exception.

Exemple simple d'utilisation d'exception

```
#include <iostream>
float div(float x, float y) throw (char *)          // throw … non obligatoire
{if (y==0.0)       throw (char *) "division par zéro";              // cast nécessaire
 return (x/y);
}
int main()
{float a,b;
  cout << "donner deux nombre réels : ";
  cin >> a;  cin >> b;
  try {       cout << "le résultat de a/b = " << div(a,b); return 1;       }
  catch (char * err) {cout << "Exception : " << err << endl;}
  return 0;
}
```

Remarque : on peut donc reprendre la main après une exception et continuer les calculs (par exemple, redemander les valeurs de a et de b).
Lorsqu'une exception est levée, on cherche un gestionnaire (traitant) d'exception en remontant la chaîne des appelants.

Au fur est à mesure que les blocs sont remontés (dans la pile des appels), les **destructeurs** des objets locaux sont appelés lorsqu'on quitte ces blocs.
Par exemple, si une fonction f() appelle la fonction g() qui lève une exception traitée au niveau de f (), en cas de levée, le bloc de la fonction g() sera quitté et ses objets locaux seront détruits.
En cas de levée d'exception, dès qu'on trouve un gestionnaire de cette exception, les autres gestionnaires de la même exception seront **ignorés**. Par exemple, si f() appelle g() est g() provoque l'erreur, si g() et f() prévoient la gestion de l'erreur, seul le gestionnaire de g() sera activé et celui de f() ne sera pas utilisé (sauf si on relève l'exception)

On peut lever plusieurs types d'exception :

```
class thing { ...};
void f()
{thing t;
  ....
  if (erreur1) throw 4;          // on lève une exception de type entier
  if (erreur2) throw t;          // on lève une exception de type thing
  ...
}
…
…
try { … }
catch (thing x) { …}
catch (int i) { … }
…
```

Un fonction peut spécifier dans son entête les types d'exceptions qu'elle peut lever :

```
class divzero              // une classe d'exception
{
   ...
}

void div(int n, int m) throw(int, float, divzero)
{
        ...
}
```

Ici, la fonction **div** spécifie qu'elle peut lever une exception de type *int*, de type *float* ou de type *divzero*. Si la fonction ne précise pas ses types d'exceptions, elle peut lever tout type d'exception.

On peut refuser à une fonction la levée de toute exception par la déclaration :

```
void div(int n, int m) throw()          // div ne peut lever aucune exception
{.....}
```

Un exemple : la pile

```
#define N 20            // nombre maximum d'élément dans la pile
class pile {
        int tab[N];
         int top;               // indice du sommet de la pile
public :
        pile() {top = -1;}
        void empiler(int e)
                {if (top < N-1) tab[++top] = e;
                 else throw (char *)  "pile pleine";
                }
        int depiler()
                {if (top > -1) return tab[top—];
                 else throw (char *)  "pile vide";
                }
};

void main() {
 pile mapile;
 int i=10, j=50;
 try   {mapile.empiler(i);
         mapile.empiler(j);
         j=mapile.depiler();
         .....
        }
 catch(char * S)        // exception simple de type char *
        {printf("erreur dans la pile : %s\n", S);
         // on peut éventuellement décider de poursuivre les opérations
        // par exemple, si la pile est pleine, on peut augmenter sa taille (après une permission)
        }
}
```

Un gestionnaire d'exception peut arrêter le programme mais il peut entreprendre d'autres actions. Dans l'exemple de la pile, on peut décider qu'en cas d'erreur "pile pleine", on peut augmenter la taille de la pile et recommencer les opération d'empiler.

```
void empiler_bis(int e)
{try {empiler(e);}
 catch(char * S)                // la seule exception possible est "pile pleine"
        {augmenter_taille();
          empiler(e);           // on recommence
        }
}
```

Dans l'exemple de la pile, on peut définir des classes d'exception et donner plus d'information sur les circonstances de la levée de l'exception.

Un autre exemple : le contrôle de débordement dans un vecteur

```
class vecteur
{int * p;
 int taille;
public :
        class debordement {};          // prévision de l'exception débordement du tableau
        int & operator[](int i);       // l'opérateur d'accès au ième élément du vecteur
        ....
};
int& vecteur::operator[](int i)
{if (0 <= i && i<taille) return p[i];  // l'indice valide
 throw  debordement();                       // Ici, l'exception est levée car l'indice est incorrect
                                       // création à la volée  d'un objet avec le constructeur vide
}
                                       // Ne pas oublier les () dans debordement()
void f(vecteur & V)
{ int ele; int j;
  try  {ele = V[j];}                   // accès au jème élément du vecteur. Erreur possible
catche (vecteur::debordement)          // L'objet reçu pourrait être nommé
        {// ici, il y a eu une erreur et l'exception débordement a été levée. On traite.
        }
}
```

Exemple complet : pile

Dans la version suivante, on crée deux classes d'exceptions **vide** et **plein** sous forme de classes imbriquées dans la pile. Lorsqu'une exception *plein* est levée, on agrandit la pile et on refait empiler.

```
#include <stdio.h>

#define N 3             // nombre minimum  d'éléments dans la pile
class pile {
        int *tab;       // le contenu est un tableau modifiable (on peut l'agrandir)
         int top;       // indice du sommet de la pile
         int max_ele;   // nbr maxi d'éléments possibles
public :
        class plein {           // classe d'exception
        public :
                int val;
                plein(int i) {val=i;}
        };

        class vide {};          // classe d'exception

        pile() {
                tab= new int[N]; max_ele=N;top = -1;
        }
        void empiler(int e){
                 if (top < max_ele-1) tab[++top] = e;
                 else throw plein(e);
        }
```

```
        int depiler() {
                if (top > -1) return tab[top--];
                else throw vide();
        }
        void agrandir()                 // réglet le manque de place
        {int * t=new int[max_ele+N];
         for (int i=0; i<max_ele; i++) t[i]=tab[i];
         delete [] tab; max_ele+=N;
         tab = t;
        }
        void empiler_bis(int e)                 // on essaie d'empiler. Si exception, on agrandit
        {try {empiler(e);}                              // la pile et on recommence
         catch (pile::plein &p)                 // sûrement l'exception  plein
                {printf("il y a eu l'exception sur %d\n",p.val);
                 agrandir();
                 empiler(e);
                }
        }
};

void main() {
 pile mapile;
 int i=10, j=50;
 try    {       mapile.empiler_bis(i);
                 mapile.empiler_bis(j);
                 mapile.empiler_bis(i);
                 mapile.empiler_bis(j);

                 j=mapile.depiler(); printf("élément %d dépilé \n",j);
        }
 catch(pile::plein &p)
        {printf("pile pleine sur %d\n", p.val);      }
 catch(pile::vide &v)                   // on ne fait rien de "v". Il peut ne pas être nommé
        {printf("pile vide\n");    }

}
```

On remarque que l'exception *pile::plein* n'arrivera jamais dans *main* car elle est traitée dans *empiler_bis*. Le bloc englobant le plus proche de cette exception est donc *empiler_bis*.

Un gestionnaire d'exception peut décider de faire remonter une exception vers les blocs supérieurs. Cette exception peut être celle qu'il a reçu ou peut être une nouvelle exception. Si le gestionnaire doit retransmettre la même exception (qu'il a reçu), il exécute **throw** sans argument.

Exemple de retransmission d'exception

```
#iclude <iostream.h>
void f(int x) {
   cout << "séquence try catch de f \n" ;
   try       {
       if (x==1) trow 1;
        if (x == 2) throw 3.14;
        cout << "fin de la séquence try catch de f \n";
   }
 catch(int v)       {cout << "exception int de f \n";}
 catch(float v)     {cout << "exception float de f \n"; throw;}          // on retransmet
}

void main(){
 int a;
 cout << "donner une valeur pour a :"; cin >> a;
 cout << " séquence try catch de main \n" ;
 try {
       f(a);        // f peut lever des exceptions
        // Arrivé ici : pas d'exception levée dans "f"
         cout << "fin de la séquence try catch de main \n";
  }
 catch(int v)       {cout << "exception int de main \n";}          // n'arrive jamais car f la traite
 catch(float v)     {cout << "exception float de main \n";}
 catch(...)         {cout << "toute autre exception de main \n"}
}
```

Dans cet exemple, la fonction f capte les exceptions de type *int* et *float*. L'exception de type *int* ne pourra donc pas arriver à *main* car elle est traitée dans *f*. Par contre, si une exception de type *float* est levée, *f* affiche un message et la retransmet à *main* par **throw**.
Comme on peut le remarquer, l'expression **catch(...)** permet de capturer toute exception.

Les exceptions sont d'une grande utilité dans l'implantation des TDA, en particulier dans le cas des fonctions partielles.

Un exemple avec reprise des calculs

Si on cherche à connaître le type de la première médaille gagnable en soulevant des poids depuis 1000 Kg en allant anté crescendo, on pourrait avoir le code suivant :

```
medaille competition (int poids) {
        if (poids > poids_max) throw (String) "trop lourd";
        soulever(poids, N);             // N = nombre de points associé au poids N soulevé
        return la_medaille_pour_ces_points(N) ;
}

void main() {
        int P=1000;       // on commence à 1000 Kg
        bool continue; medaille M;
        do {
                continue = false;
                try {  M=competition(P);      // Si on passe, pas d'eception
                        podium(M);            // le podium de M
                }
                catch(String S) {
                        if  (S=="trop lourd") { P -= 10; continue=true;}
                }
        } while(continue);

        // ici, examiner la médaille gagnée.
}
```

Exercice et TP Dico

Réaliser la classe dictionnaire de mots (Dico). Vous pouvez utiliser la classe String.
Un Dico est une table (au sens abstrait pouvant être réalisé par tableaux, listes, arbres, etc.) qui est une collection de couples (clé x info).
Dans le coupe (clé x info), la clé permet d'accéder à l'information.
Exemples de Dico : un dictionnaire anglais - français, un annuaire téléphonique, une liste d'élève et leurs notes, un tableau de livre avec leur prix, etc.

Ecrire les constructeurs, destructeur, copie constructeur et les opérateurs :

+ :	Dico x couple -> Dico
+ :	Dico x Dico -> Dico
- :	Dico x clef -/-> Dico // partielle
== :	Dico x Dico -> booléen
=! :	Dico x Dico -> booléen
= :	Dico x Dico -> Dico
<< :	Dico x flot -> flot
membre :	Dico x clef -> bool
[] :	Dico x clé -/-> info
.....	

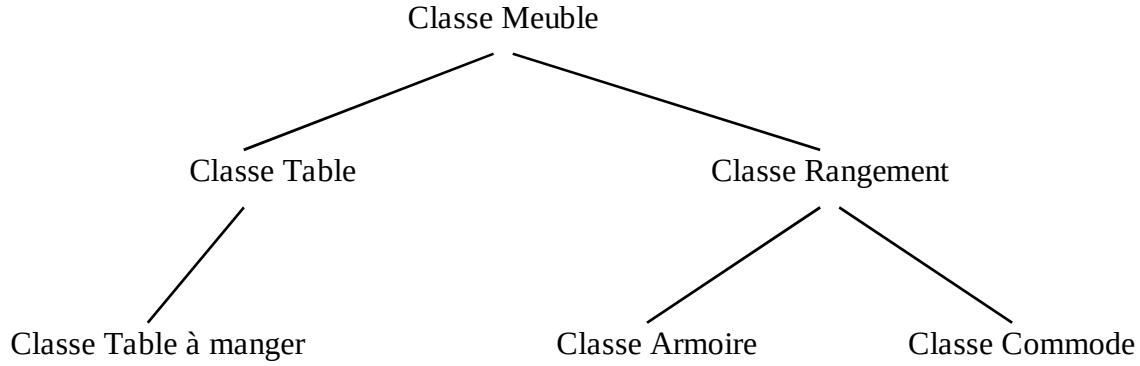
On peut éventuellement créer une classe *couple* et s'en servir dans *Dico*.

Dérivation de classes (héritage)

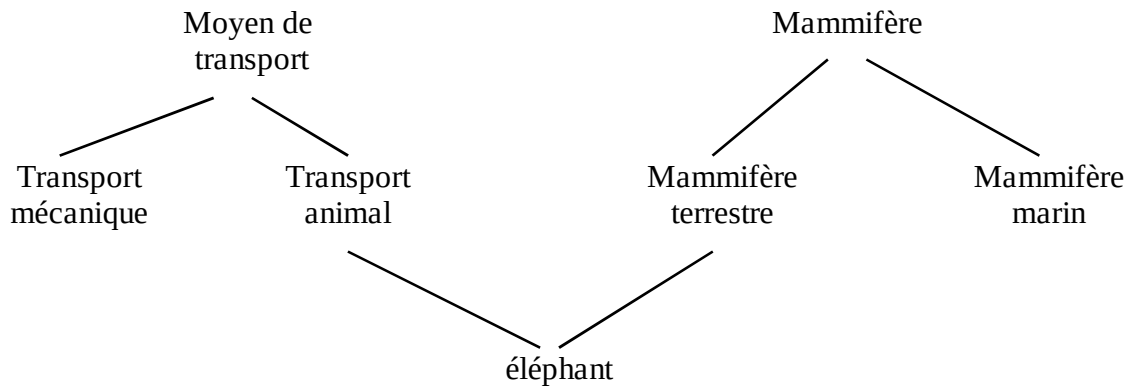
- Une classe compilée et testée est une classe réutilisable. Elle peut faire partie d'une classe (ses instances figurent dans une composition). Elle peut aussi servir de classe de base et donner naissance à d'autres classes plus spécifiques.
- Une classe compilée et testée ne doit pas être modifié (par ses utilisateurs) car d'autres utilisateurs peuvent s'en servir. Leur code dépend de la classe existante.
- Une classe compilée est dite **fermée** car on ne peut pas remettre en cause son code pour prendre en compte de nouvelles applications.
- Une classe compilée peut en même temps être **ouverte** car elle peut servir de base à la construction de nouvelles classes.

- A partir d'une classe existante A, on peut créer une classe voisine B. On ne définit pas complètement la nouvelle classe B mais on crée une classe dérivée de A en indiquant seulement les différences entre la **classe de base** A et la **classe dérivée** B.
- La réutilisation de classe par *dérivation* s'appelle **héritage**.
- La classe dérivée hérite des membres de la classe de base. Un objet instance de la classe dérivée a accès à ses propres membres ainsi que aux membres de la classe de base.
- La classe dérivée apparaît comme une copie de la classe de base avec quelques modifications.
- La définition d'une classe B dérivée d'une classe A représente la relation **sorte-de** entre B et A : on dit que *B est une sorte de A*.
- En C++, une classe peut être dérivée de plusieurs classes de base (héritage multiple).
- L'utilisation répétée de l'héritage permet de construire une hiérarchie de classes représentant une relation de spécialisation entre ces classes.

Exemple d'hiérarchie simple

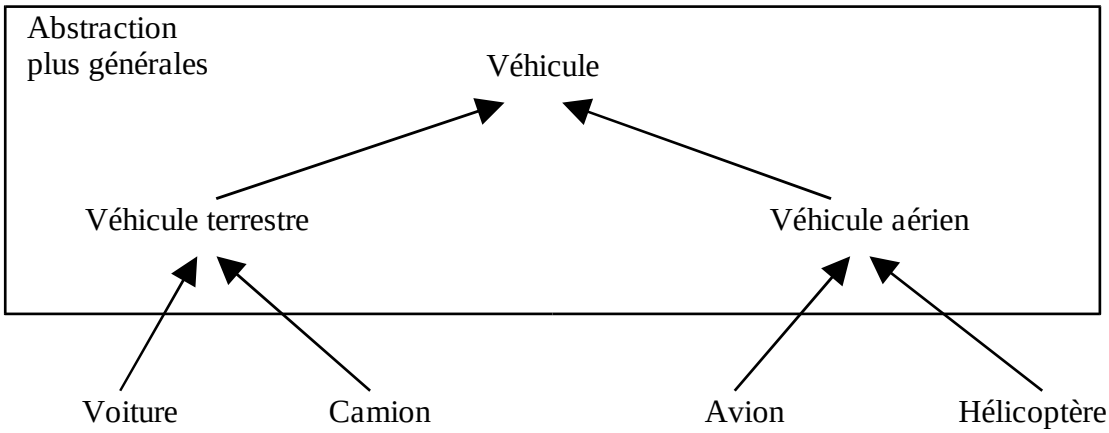


Exemple d'hiérarchie multiple

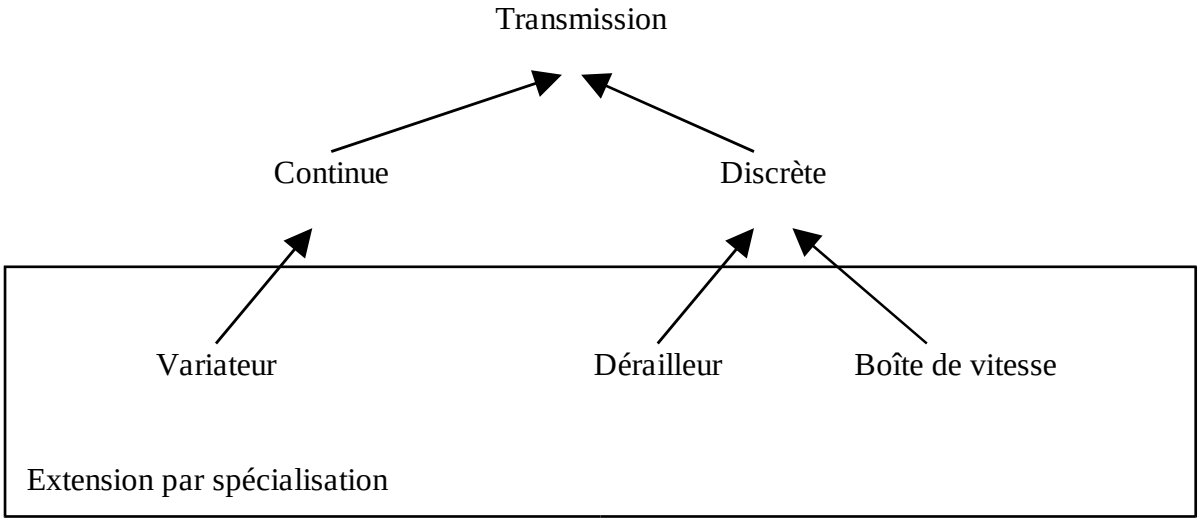


Dans la phase de spécification (avant le développement d'une application), l'héritage et la dérivation sont utilisées pour la **généralisation** ou pour la **spécialisation** de classes. Ainsi, dans un graphe d'héritage, le parcours du haut vers le bas va de la généralisation à la spécialisation alors que le parcours du bas vers le haut va de la spécialisation à la généralisation.

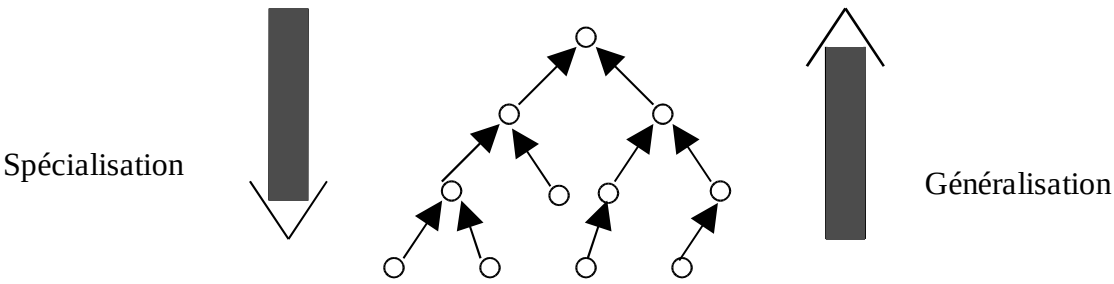
Dans une démarche de généralisation, on part des objets du monde réel bien identifiés et on essaie par abstraction de trouver les classes supérieures pour mieux ordonner et comprendre.
Exemples de généralisation à partir des objets bien connus :



Par contre, dans une spécialisation, on essaie de capturer les particularités d'un ensemble d'objets non discriminés par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par de nouvelles classes, sous classes des classes existantes. La spécialisation est une technique très efficace pour l'extension cohérente d'un ensemble de classes.
Exemple :



Graphe (arbre) de classes :



Exemple :

Soit la classe générale *livre* :

```
class livre
{String auteur;
  int nombre_de_pages;
  int prix;
public :
 livre(String A, int N) {auteur = A; nombre_de_pages = N; }
 int calcul_prix() {return prix * 1.05;}                    // 5% de TVA
 ....
};
```

Un *livre pour enfants* est une **sorte de** livre dont le prix est réduit à 90% du prix fixé. Les livres pour enfants ont une fourchette d'âges (min .. max).

```
class livre_d_enfants : public livre
{int age_mini, age_maxi;                          // fourchette d'ages
public :
 livre_d_enfants(String A, int N, int min, int max)  : livre(A, N)          // constructeur de livre
        { age_mini = min; age_maxi = max; }

  int calcul_prix()
        {int P = livre::calcul_prix();                          // calculer le prix du livre
         return (P * 0.90);
        }
   .....
};
```

Un *livre pour enseignant* est une autre **sorte de** livre avec ses particularités. Le prix d'un livre pour enseignant est le même que celui du livre en général :

```
class livre_d_enseignants: public livre
{String Discipline;
  int niveau;
public :
 livre_d_enseignants(String A, int N, String D, int M)  : livre(A, N)
        { Discipline = D; niveau = M; }
  .....
};
```

Remarque : les constructeurs, le destructeur et l'opérateur '=' de la classe de base ne sont pas hérités par la classe dérivée.

A voir plus loin :

- Le constructeur de la classe dérivée appelle le constructeur de la classe de base dans son entête.
- Une fonction de la classe dérivée peut appeler une fonction de la classe de base (sous certaines conditions qui dépendent du mode de l'héritage).

Récapitulatif d'appel de constructeurs – destructeurs dans un héritage

Règle : C++ doit initialiser les instances par des appels aux constructeurs.
Si on ne fournit pas ces constructeurs, C++ place des appels (voir ci-dessous).
Dans le cas d'héritage, il vaut mieux fournir tous les constructeurs nécessaires.

Exemple d'appel de constructeur vide

```
#include <iostream>
class base {
        int i;
public :
        base() {cout << " constructeur de Base \n";}
        ~base() {cout << " destructeur de Base \n";}
};

class derivee : public base {
        // rien pour voir les appels automatiques
};

void main() {
        derviee d;
}
```

La création de "d" appelle le constructeur vide de "base". Ce qui veut dire que le constructeur vide fourni par C++ contient un appel au constructeur vide de base.

A la suppression de "d", le destructeur de base est appelé.

On n'hérite pas de constructeur, destructeur, opérateur '=' mais C++ place un appel au constructeur vide et le destructeur de "base" dans "derivee".

Même si on déclare un constructeur vide avec un corpe {}dans "derivee", C++ place un appel au constructeur de base dans le constructeur "derivee". Même chose que dans le cas de la composition.

Même si on déclare un constructeur non vides pour "base", il faut déclarer aussi un constructeur vide dès lors que l'on compte créer "derivee d;". Dans le cas de cette création, C++ place un appel au constructeur vide de base dans le constructeur vide de "derivee"

En cas de déclaration "derivee d", si le constructeur vide de "derivee" est déclaré ainsi que divers constructeurs de "base" (sauf le constructeur vide de base), on aura un message d'erreur : absence de constructeur vide de base (qui n'est en apparence pas invoqué dans la déclaration "derivee d").

si on donne un constructeur vide pour "derivee", il faut en donner aussi à "base" surtout si une déclaration telle que "derivee d" peut exister.

Modes d'héritage

Rappel : dans une classe, on peut avoir 3 zones (sections) :
- private
- protected
- public

Exemple :

```
class truc
{private :
        int x;
 protected :
        int y;
 public :
        int z;
        void f() {...}
};
```

Les accès aux données sont régis par les règles suivantes :

Un élément de la zone **private** est — **truc U;**
- inaccessible par une instance — U.x = 35; — Interdit, x est privé
- accessible par une fonction membre
- inaccessible par **héritage** (la classe dérivée n'y a pas accès)

Un élément de la zone **protected** est — **truc U;**
- inaccessible par une instance — U.y = 35; — Interdit, y est *protected*
- accessible par une fonction membre
- potentiellement accessible par **héritage** (dépend du mode de l'héritage)

Un élément de la zone **public** est — **truc U;**
- accessible par une instance — U.z = 35; — ok
- accessible par une fonction membre
- potentiellement accessible par **héritage** (dépend du mode de l'héritage)

Les trois modes d'héritage sont possibles

private, protected , public

class ***base***
{.....};

class dérivée : ***private*** *base* *class dérivée :* ***protected*** *base* *class dérivée :* ***public*** *base*
{ .... } ; *{ .... } ;* *{ ..... };*

Les zones private, protected et public de la classe de base sont accessibles selon certaines conditions qui dépendent du mode de l'héritage.

Pour trouver le mode d'accès à un attribut X de la classe de base dans la classe dérivée, on peut procéder de la manière suivante :
Supposons une relation d'ordre représentant le degré de protection des attributs d'une classe :
private > protected > public
- soit M le mode d'héritage (M= private / protected / public)
- soit N la section dans laquelle l'attribut X de base est déclarée (N= private / protected / public)
Pour connaître le mode d'accès à X, on calcule *max(M, N).*

Remarque : dans tous les cas, un attribut privé de la classe de base est inaccessible à la classe dérivée.

Exemples :
- si X est déclaré private dans la classe de base avec un héritage quelconque :
X *inaccessible* dans la classe dérivée
- si X est déclaré publique dans la classe de base avec un héritage private :
max(private,publique)=private donc X est private dans la classe dérivée
X *accessible private* dans la classe dérivée
- si X est déclaré protected dans la classe de base avec un héritage publique :
max(protected,publique)=protected donc X est protected dans la classe dérivée

Résumé :
- *Héritage private*

classe de base	classe dérivée
zone private	**inaccessible** (car zone private de base)
zone protected	private (accessible private dans dérivée)
zone public	private ==

- *Héritage protected*

classe de base	classe dérivée
zone private	**inaccessible** (car zone private de base)
zone protected	protected
zone public	protected

- *Héritage public*

classe de base	classe dérivée
zone private	**inaccessible** (car zone private de base)
zone protected	protected
zone public	public

Exemple :

```
class base
{private :
        int Pri_X;
 protected :
        int Pro_X;
 public :
        int Pub_X;
        void aff_base() {printf("%d %d %d ", Pri_X, pro_X, Pub_X);
        vaut() {return Pri_X;}
        .....
};
```

La classe *dérivée* n'aura pas accès à *Pri_X* quelque soit le mode d'héritage.
La classe *dérivée* aura accès à *Pro_X*. Cet attribut sera private (ou protected) dans *dérivée* si le mode d'héritage est private (ou protected).
La classe *dérivée* a accès à *Pub_X*. Les instances de *base* ont accès à *Pub_X*. Par contre, si le mode d'héritage est private ou protected, les instances de *dérivée* n'auront pas accès à *Pub_X*.

```
class dérivée : public base
{...
 public :
        void aff_deriv ()
                {printf("%d %d ", Pro_X, Pub_X); }          // Pri_X inaccessible
        ....
};

void main()
{ base B;
  dérivée D;
  D.aff_deriv();
  D.aff_base();              // interdit si le mode d'héritage private/protected
  printf("%d", B.Pro_X);     // ERREUR, Pro_X est protected
  printf("%d", D.Pro_X);     // ERREUR, Pro_X est protected
}
```

Dans *main()*, on ne pourra pas appeler *D.aff_base()* si le mode d'héritage est private/protected car une fonction/donnée publique de *base* devient private/protected et donc inaccessible pour les instances.

Résumé :

Une donnée *private* de la classe de base est inaccessible à la classe dérivée et aux instances de base ou de dérivée.

En dehors de cette règle générale, on calcule le *max(mode_héritage, mode_attribut_base)* pour connaître le mode d'accès à un attribut dans la classe dérivée.

En général, on utilise dans la classe de base les zones *protected* (au lieu de *private*) si l'on veut que la classe dérivée puisse avoir accès aux données de base sans que les instances y aient accès.

Appel de constructeurs

- Pour construire une instance de dérivée, on doit construire une instance de la classe de base puis faire un traitement spécifique sur les ajouts par rapport à la classe de base.
- Dans le constructeur de la classe dérivée, on appelle d'abord le constructeur de la classe de base.

Exemple d'appel automatique des constructeurs

```
#include <iostream>
#include <string>
using namespace std ;          // nécessaire surtout dans Visual C++

class base{
 int x,y;
  public :
       base(int i, int j) {x=i; y=j;}
       //...                  // l'absence de constructeur vide pose problème (voir plus bas)
       ~base() {cout << "destructeur de base \n";}
};

class derivee : public base{
 float z;
  public :
       derivee() : z(0) {}                          // sans appel au constructeur vide de base => ERREUR
                                                    // car C++ place un appel à "base()"
       derivee(int i, int j, float k) : base(i,j)
             {z = k;}
       //....
};

void main() {
       derivee d();                  // nécessite un constructeur vide dans derivee
       derivee d(1,2,3);          // Ne pose pas de problème
}
```

Remarque : le constructeur de dérivée peut également être écrite sous la forme :

dérivée(int i, int j, float k) ***: base(i,j), z(k)*** *{}*

- L'appel du constructeur de la classe de base est situé entre l'entête et le corps du constructeur de la classe dérivée précédé par ':'
- Cet appel est identifié par le nom de la classe de base.
- On peut appeler en cascade plusieurs constructeurs en cas d'héritage multiple.
- Un constructeur de la classe dérivée peut appeler les constructeurs de ses classes de base puis appeler les constructeurs de ses membres instances d'autres classes.

- A la fin de *main*, le destructeur de "derivee" est appelé. Même si nous n'en avons pas défini dans "derivee", le message " *destructeur de base*" sera affiché, i.e. C++ place un appel au destructeur de "base" dans le destructeur qu'il fournit pour "derivee".

Accès aux fonctions membres de la classe de base

Exemple d'accès aux fonctions membres de la classe de base

```
class compteur
{int valeur;
  public :
        compteur() {valeur = 0;}
        void inc() {if (valeur < 32767) valeur++;}
        void dec() {if (valeur > 0) valeur--;}
        int vaut() {return valeur;}
};
class compteur_contrôle : public compteur
{int max_val;
  public :
        compteur_contrôle(int max) : compteur()          // appel du constructeur de base
              {max_val = max;}
        void inc()                                       // redéfinition de fonction membre
              {if (vaut() < max_val) compteur::inc();}   // appel d'inc de base
};
```

Ici, la classe dérivée redéfinit inc. *inc* de la classe dérivée appelle *inc* de la classe de base.

Exemple : classe Etudiant

La classe *Personne* est définie puis la classe *Etudiant* est dérivée de la classe Personne.
On utilise la classe String (composition).

Exemple de la classe Etudiant

```
#include "String.hpp"
class Personne
{String nom;
  int No_SS;
  public :
        Personne() : nom(""), No_SS(0) {}          // appel du constructeur de String
        Personne(String  S, int N) : nom(S), No_SS(N) {}
        void edit() {cout << "nom = " << nom << "No SS = " << No_SS << endl;}
        ....
};

class Etudiant : public Personne{        // un Etudiant est une sorte de Personne
 String Université;
  int annee;
  public :
        Etudiant(String N, int I, String U, int A) : Personne(N, I)
              {Université = U; annee = A;}
        Etudiant(String N, int I) : Personne(N, I) , Université(""), annee(0) {}
        void edit()
              {Personne::edit();          // appel d'edit() de base
               cout << Université << annee << endl;
              }
        ....
};
```

Modifications par rapport à une classe de base

Une classe dérivée est définie relativement à une classe de base. Elle peut introduire trois types de différences par rapport à la classe de base :

- ajout de fonctions membres
- modification de fonctions membres
- suppression de fonctions membres

Ajout de fonctions membres :

S'effectue en définissant ces fonctions dans la classe dérivée

Modification de fonctions membres :

S'effectue en redéfinissant dans la classe dérivée des fonctions de la classe de base.

Exemple :

```
class base                                  class dérivée : public base
{....                                       {.....
 public :                                    public :
      void f0() {...}                             void f2() { ... }
      void f1() {...}                             void f1() { ... }
};                                          };

void main()
{base A;
  dérivée B;
  A.f0();          // appel de f0() de base
  B.f0();          // appel de f0() de base
  A.f1();          // appel de f1() de base
  B.f1();          // appel de f1() de dérivée (modification)
  B.f2();          // appel de f2() de dérivée (nouvelle fonction)
  A.f2();          // ERREUR, n'existe pas
}
```

Suppression de fonctions membres :

- On définit un héritage private : ceci masque toutes les données et fonctions de la classe de base;
- On précise explicitement les fonctions qui doivent être conservées dans la classe dérivée.

Exemple :

```
class base
{int i,j;
  public :
      bas(int ii = 0, int jj = 0) {i=ii; j=jj;}          // valeurs par défaut
      void init(int val) {i=j=val;}
      void annuler() {i=j=0;}
};
class dérivée : base                  // par défaut, héritage private.Tout devient private
{public :
      dérivée(int x, int y) : base(i, j) {}          // appel du constructeur de base
      void init(int val)
            {base::init(val);}        // appel explicite à init de base. Base::init est devenue
};                                    // private dans la classe dérivée
```

```
void main()
{base A;
 dérivée B(1, 2);
 A.init(1);        // appel de la fonction publique init de base
 A.annuler();      //      ==          ==          annuler de base
 B.init(2);        // appel de fonction publique init de dérivée
 B.annuler();      // ERREUR. annuler est devenue private et inaccessible aux instances
}
```

Remarques :

Seule *init* est accessible par B

Dans la classe dérivée, on peut remplacer *void init(int val) {base::init(val);}*

par **base::init;**

Ce qui a pour effet de préciser qu'*init* de dérivée = *init* de base.

Syntaxe des appels aux fonctions de la classe parent

Soit la classe Personne définie avec un opérateur '=' et l'opérateur '<<'. La classe Etudiant dérive de la classe *Personne*.

Exemple d'appel des fonctions de la classe parent

```
#include "Personne.hpp"

class Etudiant : public Personne  {      // un Etudiant est une sorte de Personne
      String dept;
      int annee;
 public :
      Etudiant() : Personne() , dept(), annee(0) {}
      Etudiant(String N, int I, String d, int A) : Personne(N, I)
            {dept = d; annee = A;}
      Etudiant(String N, int I) : Personne(N, I) , dept(""), annee(0) {}
      Etudiant(const Etudiant & E) : Personne(E) {dept=E.dept; annee =E.annee; }
      Etudiant operator=(const Etudiant &E) {
            if (this == &E) return *this;
            this->~Etudiant();      // la présence de this est importante ici
            Personne::operator=(E);     // et non : *this=E       => boucle infinie
            dept=E.dept; annee =E.annee;
            return *this;
      }

      friend ostream& operator<<(ostream & f, const Etudiant &E) {
            f << (Personne &) E;          // Personne::operator<<(f, (Personne) E) ne passe
                                          // pas  car '<<' est friend dans Personne.
            f << E.dept << " " << E.annee << endl;
            return f;
      }
      ....
};
```

Gestion des objets dynamiques

Objets statiques ou dynamiques

Objets statiques :
- Les objets déclarés statiquement sont totalement déterminés. Leur création ne dépend pas d'un résultat intermédiaire. Par exemples, les instances de classes faisant partie d'une autre classe, celles déclarées globales dans le programme ou locales à une fonction sont crées lors du lancement du programme ou lors de l'appel de la fonction.
- Les objets déterminés à la compilation ont une durée de vie analogue à celle d'une variable d'un type de base.
- Un objet local à un bloc disparaît à la fin du bloc par un appel à son destructeur.

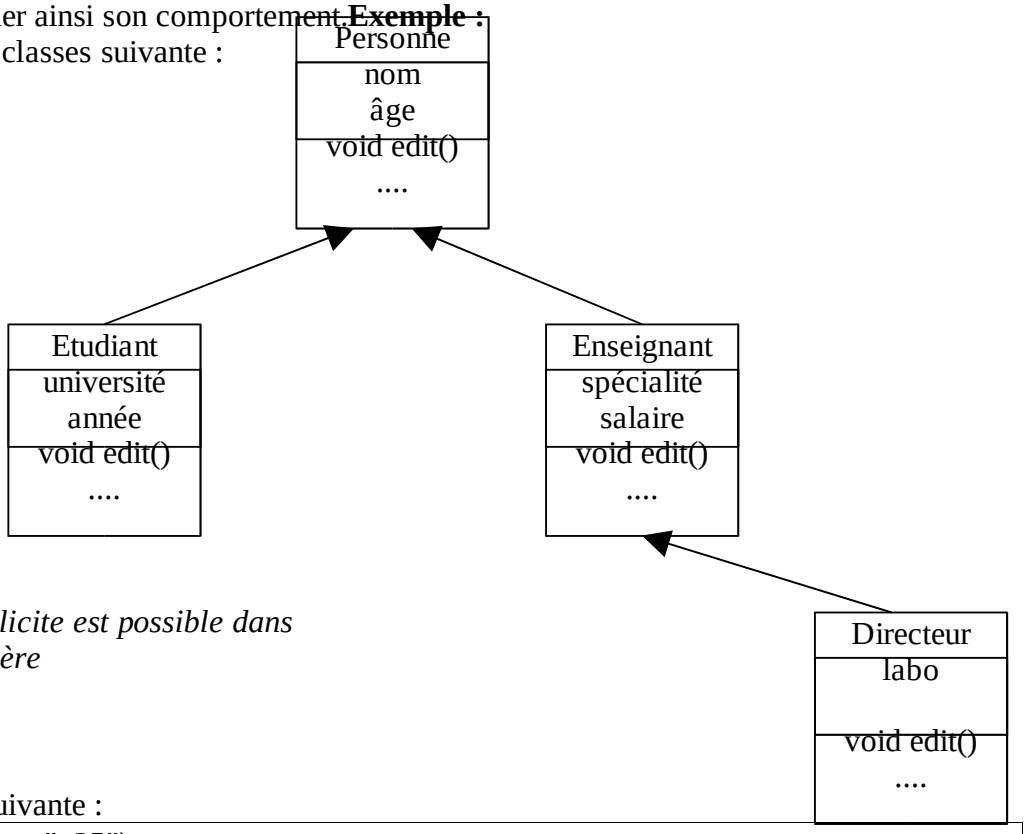
Objets dynamiques :
- Données sans nom accessibles par pointeurs.
- Opérateurs *new* pour la création et *delete* pour la suppression.
- Création et libération pendant l'exécution dans le tas (heap).

Exemple d'allocation dynamique

```
class stock {
        char * nom;
         int q;
public :
        stock() {...}
        stock(char * n, int i)  { .... }
        stock(const stock &s) { ... }
        ~stock() {.....}
        ...
};

void main() {
 stock S1("tomates", 5000);
  stock * Ptr1, * Ptr2, * Ptr3 ;
  Ptr1 = new stock;
  Ptr2 = new stock(S1);
  Ptr3 = new stock[20];              // nécessite le constructeur vide de stock
  ...
  delete Ptr1; delete Ptr2;
  delete [] Ptr3;
}
```

Ici, *S1* est un objet local à main. Il est créé dès le lancement du programme. Il disparaît à la fin du *main*. Il est complètement déterminé à la compilation.
Ptr1 et Ptr2 sont des pointeurs sur *stock*. Ptr1 est créé avec le constructeur vide de *stock*. Ptr2 est créé avec le constructeur par copie. Les objets pointés par Ptr1 et Ptr2 sont créés pendant l'exécution et ne seront supprimés que par *delete*.

En C++, il est possible d'intervenir sur un objet pointé par un pointeur, de changer l'objet pointé par un autre et de modifier ainsi son comportement.**Exemple :**
Soit la hiérarchie de classes suivante :



La conversion implicite est possible dans le sens fille vers mère

Avec l'application suivante :

```
Personne P("jean", 25");
P.edit();                          // affichage des données de la Personne P
Etudiant E("pierre",26, "Galata", 4);
E.edit();                          // affichage des données de l'Etudiant E
P = E;                             // appel de '=' de Personne. Possible; E perd
                                   // ses données pour devenir une Personne
P.edit();
```

P.edit() affiche les données de la **Personne P** car la variable P a été associée une fois pour toutes à la classe Personne. A noter que l'affectation *P=E* est acceptée car la classe Personne est plus générale que la classe Etudiant : un Etudiant est une Personne + des attributs propres à Etudiant.

```
E = P;          // ERREUR;  sauf si l'opérateur '=' est défini dans Etudiant
```

Sans la surcharge de l'opérateur '=' dans Etudiant, cette opération ne passe pas la compilation (g++) car la conversion directe d'une Personne en un Etudiant est illégale (l'erreur syntaxique viendra de l'absence de *operator=(Personne))*. Cependant, même si la compilation réussit (avec '=' défini dans Etudiant), l'exécution risque d'être avortée car une personne ne devient pas un Etudiant.

```
Personne * P1 = new Personne("Jacques", 24);
P1 -> edit();                  // appel de la fonction edit() de Personne
Etudiant * E1 = new Etudiant("marie", 23, "Grenoble", 3);
E1 -> edit();                  // appel de la fonction edit() de la classe Etudiant
```

Si on écrit :

```
P1 = E1; P1 -> edit();
```

Ici, la fonction edit() de la classe Personne est appelée et **seule** la partie Personne de l'Etudiant pointé par E1 est éditée (avec ou sans '='(Personne).

Avec *P1=E1*, la fonction edit() reste celle de la classe Personne mais les attributs de la Personne pointée par P1 sont remplacés par ceux de l'Etudiant pointé par E1. Ce comportement dynamique est réalisé grâce à l'utilisation des pointeurs.

Si l'on ajoute :

```
Etudiant * E2 = (Etudiant *) P1; E2 -> edit();
```

Cette fois, toutes les données de l'Etudiant pointé par E1 seront éditées et la fonction edit() sera celle de la classe Etudiant. Cette adaptation dynamique est réalisée grâce aux pointeurs et à la conversion de l'objet pointé par P1 en un Etudiant.

Il en est de même pour la classe Enseignant et la classe Directeur.

Supposons maintenant qu'une université désire gérer son personnel enseignant dont certains sont des directeurs de laboratoires. Pour cela, on peut envisager un tableau T qui contiendra ce personnel.
Il semble évident que le tableau T contenant le personnel doit être un tableau d'Enseignants. Ainsi, certaines cellules de ce tableau pourront contenir des Directeurs.
Il en de même pour les autres sous classes de cette hiérarchie : l'université pourrait stocker les Etudiants et les Enseignants dans deux tableaux différents ou bien les mettre tous dans un tableau de Personne car la classe Personne représente la partie commune des autres sous classes.

On a :

```
Enseignant T[N];          // déclaration du tableau T
```

Insérons dans T deux enseignants :

```
T[1] = E1; T[2] = E2;     // E1 et E2 sont des  enseignants
```

Si une cellule de ce tableau doit contenir un Directeur D, on écrira par exemple :

```
T[3] = D;
```

Mais nous avons vu plus haut que l'affectation d'un Directeur dans un Enseignant ne permet pas la conversion souhaitée et provoque une perte de données.

Si le tableau T est un tableau de pointeurs sur Enseignants, le problème n'est que partiellement réglé car comme dans les exemples plus hauts, l'appel de la fonction edit() pour chaque objet pointé par ce tableau ne permet pas d'afficher les données complètes des Directeurs (la fonction edit() reste celle de la classe Enseignant et affichera seulement la partie Enseignant des données du Directeur).

Pour pouvoir appeler la fonction edit() de la classe Directeur pour l'objet Directeur pointé par T[3], il nous faudra une conversion de la forme :

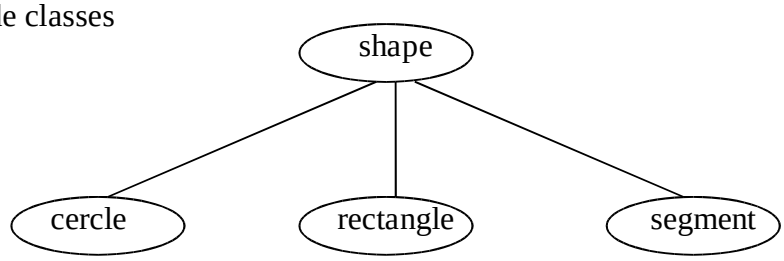
```
Directeur * D1 = (Directeur *) T[3];  D1->edit();
```

Mais pour cela, il faut a priori savoir que T[3] contient un pointeur sur un Directeur. Cette information peut être fournie par un champ contenant une indication sur la classe de l'objet pointé par chaque cellule du tableau T. Une meilleure solution est l'utilisation de "virtual".

Les pages suivantes exposent ce problème dit du **polymorphisme** et expliquent les solutions possibles à en C++.

Le Polymorphisme et fonctions Virtuelles

Un objet polymorphe est un objet susceptible de prendre plusieurs formes pendant l'exécution.
Si on a la hiérarchie de classes



Les classes *cercle, rectangle* et *segment* sont dérivées de *shape*. Ce sont des **sortes** de *shape*.
Dans le cas des objets statiques, une instance statique est déterminée à la compilation et ses fonctions membres sont disponibles et on ne peut pas les modifier. Il n'est pas possible que pendant l'exécution, un objet *shape* devienne un cercle.

Par contre, avec les objets dynamiques, un objet instance de la classe shape peut, durant l'exécution, prendre la forme d'un cercle, d'un rectangle ou d'un segment (par affectation). C'est ce qu'on appelle le **polymorphisme**.

Le problème principal dans le polymorphisme est la sélection des fonctions membres correspondant à l'objet au moment de l'appel.
Supposons que chacune de ces classes a une méthode *dessiner*. Si un objet de la classe *shape* prend la forme d'un *cercle*, comment peut on faire en sorte que la bonne méthode *dessiner* soit activée.

Mise en œuvre du polymorphisme en C++

En C++, le polymorphisme ne s'applique qu'à des objets dynamiques et par l'intermédiaire des fonctions **virtuelles**.
Le polymorphisme s'applique dans le cas des hiérarchies d'objets.

```
class shape                                   class cercle : public shape
{...                                          { ...
  public :                                      public :
      void dessiner() { ... }                       void dessiner() { ... }
};                                            };

class rectangle : public shape                class segment : public shape
{ ...                                         { ...
  public :                                      public :
      void dessiner() { ... }                       void dessiner() { ... }
};
                                        };
void main(){
  shape * pt_shape;
  cercle * pt_cercle;
  rectangle * pt_rect;
  segment * pt_seg;
   pt_rect = new rectangle(...);
   pt_shape = pt_rect;          // ici, pt_shape un objet polymorphe prend la forme d'un rectangle
   pt_shape->dessiner();
}
```

Ici, l'appel *pt_shape->dessiner();* est traduit par le compilateur en un appel à la fonction *dessiner()* de la classe *shape* (seule la partie shape du rectangle est dessinée) car dans main, *pt_shape* est déclaré pointeur sur un objet *shape*.

Or, ce n'est pas le comportement souhaité car après l'affectation *pt_shape = pt_rect;* pt_shape pointe sur un rectangle. Il faudra que la méthode dessiner() de rectangle soit appelée.

Ce mauvais choix de la fonction membre *dessiner()* est dû à la liaison statique effectuée lors de la compilation entre pt_shape et la fonction dessiner() de shape. On appelle ce mécanisme **early binding**.

Il faut mettre en ouvre une liaison dynamique entre un objet et une fonction membre pendant l'exécution (appelée **late binding**).
Dans le cas de liaison dynamique, la fonction membre appelée sera celle correspondant à la forme de l'objet à cet instant.

Dans la cas de l'exemple ci-dessus, pt_shape est déclaré pointeur sur shape et de ce fait, il a accès aux fonctions membres de shape. Mais après l'affectation *pt_shape = pt_rect*, il faudra que pt_shape change de fonctions membres et pointe celles de rectangle.

Solutions :
1-1 : Une première solution envisagée pour appeler la bonne fonction *dessiner* serait de connaître la classe d'un objet instance.
Pour cela, on peut prévoir dans la classe de base (ici *shape*), un attribut "type" qui permet de savoir quel est le type de l'objet.
Dans les constructeurs des classes, on donne la valeur de la classe au champ "type".
1-2 : Alternativement, on peut prévoir une fonction qui renvoie le type (un string) de l'objet.
Dans les deux cas, la connaissance du type de l'objet permet d'appeler la fonction adéquate.
Ainsi, dans l'exemple précédent, on interroge le "type" de l'objet dans la fonction dont le rôle est de dessiner un objet selon son type.

Cas 1-1 :

```
enum type_forme {shape, cercle, rectangle, segment};

class shape
{....
  public :
        type_forme type;
        shape(...) {type = shape; ....}
        void dessiner() { ... }
};
```

```
        class cercle : public shape
{ ....
  public :
        cercle(...) {type=cercle; ..... }
        void dessiner()  { .... }
        .....
};
```

Cas 1-2 :

```
class shape
{....
  public :
        shape(...) {....}
        void dessiner() { ... }
        string nom() {return "shape"; }
};
```

```
        class cercle : public shape
{ ....
  public :
        cercle(...) {.... }
        void dessiner()  { .... }
        string nom() {return "cercle";}
};
```

On déclare les classes *rectangle* et *segment* de la même manière.
On prévoit de plus une fonction qui décide de dessiner chaque forme géométrique en fonction de son type (nom). On remarque la conversion de type de pointeur dans cette fonction.

La fonction ci-dessous est une fonction externe. Elle est accessible par toutes les classes.

```
void print_shape(const shape * S)              // dessiner la forme selon son type
{switche (S->type)
      { case shape :          S ->dessiner(); break;
        case cercle :         cercle * C = (cercle *) S;          // conversion de shape en cercle
                              C -> dessiner(); break;
        case rectangle :      // même chose pour rectangle
        case segment :        // idem
      }
}
```

Dans le cas de la solution 1-2, on peut interroger le nom de l'objet pour décider d'appeler la fonction *dessiner* adéquate.
L'inconvénient de ces solutions est que si l'on ajoute d'autres classes à la hiérarchie de classes des objets géométriques, il faudra prévoir le type (nom) des nouvelles classes ajoutées, de modifier et de le recompiler les fichiers correspondants

.
Une solution plus élégante et proche de celle ci-dessus est envisageable dans les versions récentes du compilateur C++. Dans ces versions récentes, on a la possibilité de connaître le type (la classe) d'un objet (cf. typeid et typeinfo.h).

La solution idéal préconisée en C++ est l'utilisation des fonctions **virtuelles**.

Fonctions virtuelles

- Elles permettent la mise en œuvre de la liaison dynamique entre un pointeur et les méthodes de l'objet pointé.
- Dans la classe de base dont dérivent les différentes formes d'un objet polymorphe, on déclare **virtual** les fonctions membres pour lesquelles on veut réaliser une liaison dynamique.

Remarque : même s'il y a plusieurs niveaux de dérivation, il suffit de déclarer virtuelle la fonction correspondant dans la classe de base. Il n'est pas nécessaire de déclarer virtuelle la même fonction dans les dérivations successives.

```
class shape       {                          class cercle : public shape
      ...                                    { ...
 public :                                     public :
      virtual void dessiner() { }                 void dessiner() { ... }
      virtual void deplacer() { ..}               void deplacer() { ..}
};                                           };
class rectangle : public shape               class segment : public shape
{ ...                                        { ...
  public :                                    public :
      void dessiner() { ... }                     void dessiner() { ... }
      void deplacer() { ..}                       void deplacer() { ..}
};                                           };
void main()
{ shape * pt_shape;
  cercle * pt_cercle;
  rectangle * pt_rect;
  segment * pt_seg;
  pt_rect = new rectangle(...);
  pt_shape = pt_rect;
  pt_shape->dessiner();          // appel de la fonction dessiner() de la classe rectangle
}
```

Ici, on a une liaison dynamique (réalisée à l'exécution) et la fonction dessiner() appelée est celle définie dans la classe rectangle.

Les fonctions virtuelles doivent être définies dans la classe de base (définition éventuellement vide). Ces fonctions doivent avoir le même prototype dans la classe de base et dans la classe dérivée.

Exemple Employé-Manager

On remarque que le champ type disparaît et la fonction *print()* devient **virtual**. Dans la fonction *print_liste()*, l'appel *el->print()* activera la bonne fonction *print()* de chaque objet.

Exemple d'utilisation des fonctions virtuelles

```
#include <iostream.h>
#include "string.h"

class employe
{protected :        // pour avoir accès à nom dans le destructeur de Manager
        char * nom;
         short dept;
public :
         employe * next;
         employe(char * n, int d);
         virtual void print() const;                    // fonction virtual
         virtual ~employe() {cout <<"Destructeur d'Employé "<< nom<<"\n"; delete[] nom;}
};

class manager : public employe
{       employe * groupe;
        short level;
 public:
         manager(char * n, int l, int d);
        void print() const;
        ~manager()
        {cout <<"Destructeur de Manager " << nom;
         cout <<" : appel automatique de destructeur d'Employé \n";
         delete groupe;}                // INUTILE d'appeler le destructeur d'Employé pour
};                                      // enlever la partie haute

                        // Il faut appeler "delete groupe" pour supprimer proprement
                        // l'employé pointé par groupe car l'appel de "~employe()" ne suffit pas

employe::employe(char * n, int d) : dept(d)
{nom=new char[strlen(n)+1]; strcpy(nom,n);}

void employe::print() const
{cout << " employe : "<< nom << "," << dept << endl; }

void manager::print() const
{cout << "manager : ";
 employe::print();
 cout << "level : " << level << endl;
}

manager::manager(char *n, int l, int d) : employe(n,d), level(l), groupe(0) {}

void print_liste(const employe * el)
{for (;el;el=el->next) el->print(); }
```

```
void main()
{employe * liste=NULL;
 employe e("J.Brown", 1234);
 e.next=liste;  liste=&e;                    // chaînage

 manager m("J.Smith", 2, 1234);
 m.next=liste; liste=&m;

 print_liste(liste);

 employe * e1=new employe("M. Jack", 2341);
 e1 -> print();
  cout << endl;                              // appel de print() d'Employé

 delete e1;                                  // appel du destructeur d'Employé

 manager * m1=new manager("M. toto", 2341,200);
 e1=m1;
 e1 -> print();
 cout << endl;                               // appel de print() de Manager

 delete e1;                                  // appel du destructeur de Manager
 e1 = &m;
 e1 -> print(); cout << endl;                // appel de print() de Manager
}
```

Dans les dernières lignes de main(), on remarque l'intérêt des fonctions virtuelles dans l'adaptation automatique et dynamiques des fonctions membres des classes.

Rappelons que grâce à *virtual*, la conversion des pointeurs est devenue inutile (cf. *e1=&m* ci-dessus).

Remarque sur les destructeurs : Ici, le destructeur de la classe Manager appelle automatiquement le destructeur de la classe Employé.

Trace de l'exécution :
Les affichages à l'écran des messages de cet exemple (appel de print et messages de destruction) sont les suivants :

```
Manager : Employé : J.Smith,1234 level : 2
Employé : J.Brown,1234                    // Fin print_liste

Employé : M. Jack,2341
Destructeur d'Emplye M. Jack
Manager : Employé : M. toto,200 level : 2341
Destructeur de Manager M. toto : appel automatique de destructeur d'Employé
Destructeur d'Employé  M. toto
Manager : Employé : J.Smith,1234 level : 2    // Fin corps main
                                              // début des suppressions des variables
Destructeur de Manager J.Smith : appel automatique de destructeur d'Employé
Destructeur d'Employé J.Smith
Destructeur d'Employé J.Brown
```

Un autre exemple de fonctions virtuelles

Comme nous l'avons vu plus haut, l'utilisation d'un type pour connaître la classe d'un objet pour appeler la bonne fonction est une solution lourde.
Le polymorphisme est un moyen d'attribuer un nom à une action exécutée par des objets similaires, avec chaque implantation de l'action d'une manière propre à chaque objet. En C++, le polymorphisme se réalise avec *virtual*.
Les fonctions virtuelles permettent de supplanter des fonctions de la classe de base.
Pour les fonctions virtuelles, c'est le type de l'objet pointé qui détermine la fonction à appeler.

Dans l'exemple simplifié suivant, la classe de base est la classe *Employé* et la classe *Employé_étranger* hérite d'Employé. Les fonctions héritées sont supplantées (redéfinies) par les fonctions virtuelles.

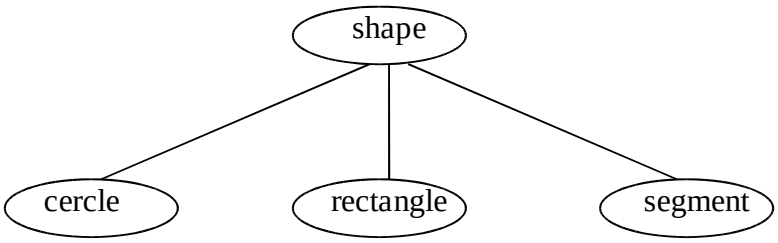
Un autre exemple des fonction virtuelles

```
#include <iostream.h>
#include "String.hpp"

class Employe
{ String nom;
  String departement;
  String poste;
  int salaire;
  public:
  Employe () :  nom(""),  salaire(0),  departement(""),  poste("")    { }
  Employe (  String n,  int sal,  String dept,  String p  )
              :  nom (n),  salaire (sal), departement (dept),  poste (p)
  { }

  virtual ~ Employe ( ) { }
  virtual void edit ()
       {      cout << "\Nom : " << nom << "\n";
              cout << "Departement : " << departement<< "\n";
              cout << "Poste : " << poste<< "\n";
              cout << "Salaire : " << salaire<< "\n";
       }
};

class Emp_etranger  : public Employe
{String pays;
  public:
  Emp_etranger () :  Employe(), pays("")   { }
  Emp_etranger (String apays  ) :  Employe(), pays (apays)  { }
  Emp_etranger (String n, int sal, String dept, String pst ,String pys )
               :   Employe(n, sal, dept, pst), pays (pys)
   { }

   ~ Emp_etranger ( ) { }

  void edit()
       {      Employe::edit();
              cout << "Pays : " << pays<< "\n";
       }
};

// Demonstration du Polymorphisme
// pour le bon dispatch d'obj dans la fonction print, il faut VIRTUAL sur EDIT d'Employe
void print(Employe * obj)
       { obj -> edit();}
```

```
void main()
{
 Employe *Employe1=new Employe("jean",10000,"ventes","chef");
 Employe1->edit();
 Employe *Employe2=new Employe("pierre",12000,"pub","adjoint");
 Employe2->edit();

 Emp_etranger *Employe3=
       new Emp_etranger("marie",1800,"market","secret","Turquie");
 Employe3->edit();

 Employe1=Employe3; Employe1->edit();      // utilisation de Virtual
Emp_etranger *Employe4=
       new Emp_etranger("helene",1700,"atelier","directrice","France");
 print(Employe2);
 print(Employe4);
}
```

Méthodes virtuelles pures et classes abstraites

Comme on l'a vu dans les exemples précédents, les méthodes virtuelles ont une implantation (éventuellement vide notée par {}) et la classe de base peut avoir des instances.
Par exemple, dans le cas des formes géométriques, nous avions la hiérarchie suivante:



On pourrait envisager dans cette hiérarchie que la classe de base *shape* regroupe les propriétés communes des formes géométriques. Mais elle ne représente aucun objet géométrique. A ce titre, c'est une classe abstraite qui servira de dérivation. Il n'est donc pas utile ni correct d'avoir des instances de la classe shape. Une telle classe est appelée une **classe abstraite**.

Une classe abstraite est une classe sans instance qui ne sert qu'à la dérivation d'autres classes.

Pour avoir une classe abstraite, il faut dans cette classe avoir au moins une fonction virtuelle appelée **fonction virtuelle pure**. Une fonction virtuelle pure est de la forme (dans la classe shape) :

```
virtual void dessiner() = 0;
```

Une telle classe ne peut pas avoir d'instance et sert seulement à la dérivation.
La méthode virtuelle pure doit être redéfinie dans les classes dérivées. **Une seule** fonction virtuelle pure suffit pour construire une classe abstraite.

On définira ensuite les classes cercle, rectangle et segment par dérivation de la classe shape. Ces classes auront des objets instances représentant des entités du monde réel.

Spécification

Pour spécifier l'architecture d'une application réalisée par une méthode orientée objets, il existe plusieurs formalismes graphiques tels que OOA (Object Oriented Analysis), OMT (Object Modeling Technique), UML (Unified Modeling Language), Booch (modèle de Grady Booch), HOOD, ... La méthode UML est un sur ensemble des méthodes Booch et OMT.

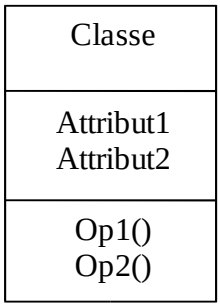
L'élément de base de l'activité de spécification dans toutes ses méthodes est la distinction et la reconnaissance des objets / classes (et de leurs propriétés).

Ces formalismes sont très proches les uns des autres, en particulier OOA, OMT et UML. Sans nous intéresser à ces méthodes en détails (voir les autres cours), nous utiliserons le formalisme graphique UML pour la spécification des applications à réaliser.

Le formalisme UML

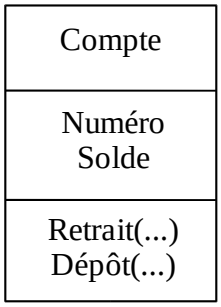
Ce formalisme est très proche du formalisme OMT.

Classe et objet



un objet est le résultat de l'instanciation d'une classe. Cette instanciation définit la relation **"est-un".**

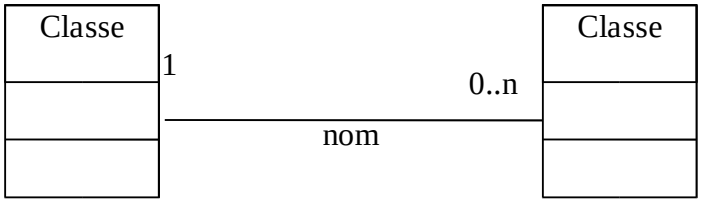
Exemple :



NB : une classe **abstraite** est une classe non instanciable; elle est utilisée seulement pour l'héritage.

Association

Une association représente un lien structurel entre classes d'objets. La plupart des associations sont binaires (connectent 2 classes).



Une association est proche d'une agrégation (voir plus loin) mais d'une sémantique plus faible (l'agrégation est une des méthodes d'organisation dominantes de la pensée humaine).
Les associations s'ajoutent aux attributs pour donner les liaisons requises demandées par un objet.

Nom de l'association : la liaison entre les classes peut être nommée.
Multiplicité : chaque classe peut participer à l'association en de 0 à N exemplaires.

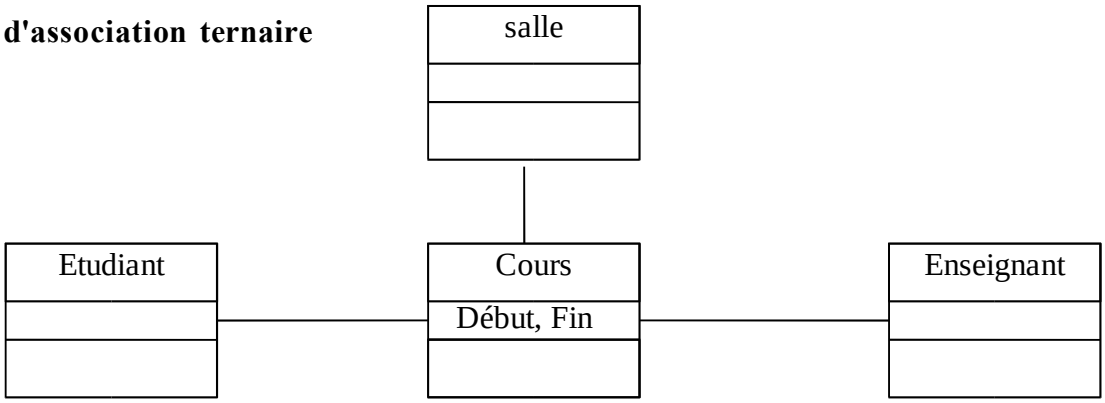
Exemple d'association

Ici, un plan de vol particulier doit être exécuté par exactement un *Avion*; tout *Avion* peut exécuter zéro ou plusieurs plans de vol.

Remarque : différence par rapport à une agrégation :
Dans une agrégation, une des classes joue un rôle prédominant par rapport à l'autre (cf. notion de composé - composante).
Dans une agrégation,, en général, un seul des rôles est important. Par exemple, le lien entre une banque eu ses comptes. On ne met pas le nom de la banque dans les comptes. Ceci est traduit par une relation orientée (flèche) de la banque vers le compte.

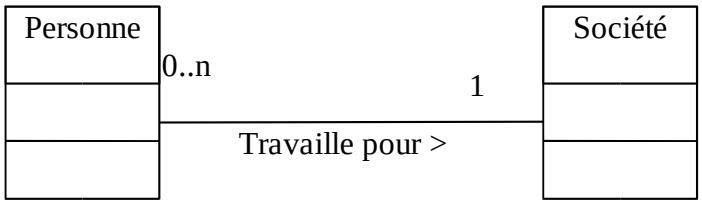
La plupart des associations sont binaires mais on peut avoir besoin d'associations n-aires.

Exemple d'association ternaire



Nommage des associations

On peut nommer une association en général par une forme verbale (telle que *travaille pour*, *est employé par*, ...):

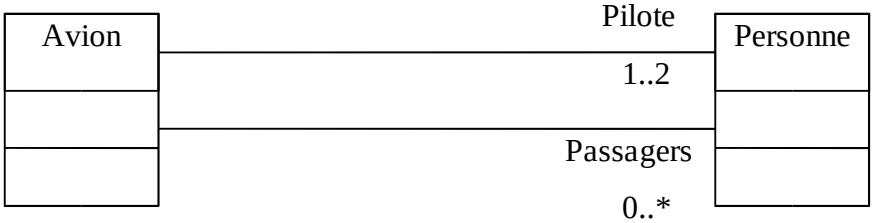


Le sens de lecture du nom peut être précisé par les signets '<' ou '>' ou par un triangle.
Le nom d'une association permet une clarification de lecture du diagramme et n'apparaît pas dans le code généré pour le diagramme.

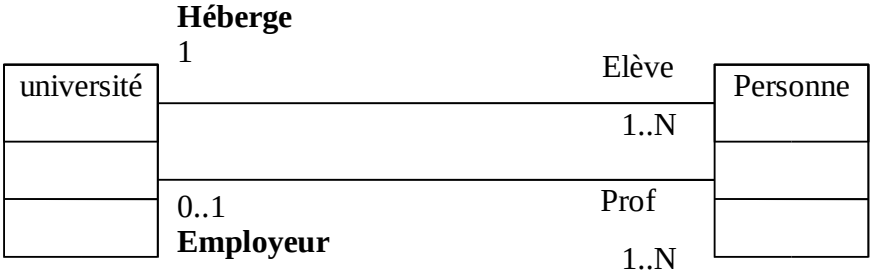
Rôles et nommage des rôles

L'extrémité d'une association est appelée **rôle**. Chaque association binaire possède deux rôles. Le rôle décrit comment une classe voit une autre classe au travers une association. Un rôle est nommé au moyen d'une forme nominale. Le nom d'un rôle se distingue du nom de l'association.
Dans une association ou agrégation, on peut attribuer un rôle (un nom identifiant) à chaque extrémité de la liaison.

En général, on ne mélange pas l'usage du nom de l'association avec le nommage des rôles. On préfère le nommage des rôles au nom de l'association, en particulier lorsque deux classes sont reliées par plusieurs associations.



Un autre exemple :



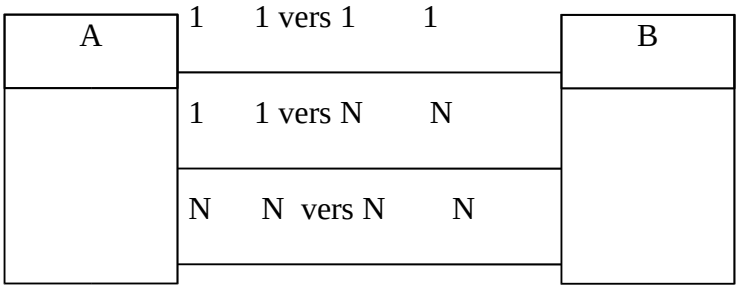
Multiplicité des associations

Chaque rôle d'une association porte une indication de **multiplicité** qui montre combien d'objets de la classes considérée peuvent être liés à un objet de l'autre classe. La multiplicité est une information portée par le rôle sous la forme de :

1 : un et un seul (valeur par défaut)
0..1 : zéro ou un
M..N : de M à N (entiers naturels)
* : de zéro à plusieurs
0..* : idem (noté également 0..N)
1..* : d'un à plusieurs (ou 1..N)

Exemple : Voir exemples ci-dessus.

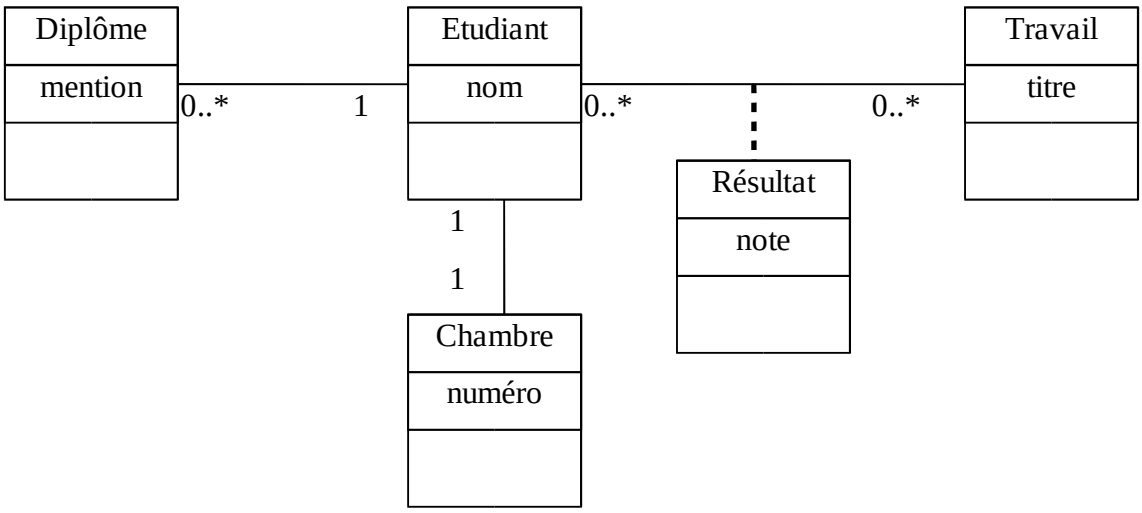
Les valeurs de multiplicité sont souvent employées pour décrire de manière graphique les associations. Les formes les plus courantes sont les associations *1 vers 1*, *1 vers N* et *N vers N*.



Classe - association (attribut de lien)

Pour les associations 1 à 1, les attributs de l'association peuvent toujours être déplacés dans une des classes qui participent à l'association. Dans le cas d'une association 1 vers N, le déplacement est généralement possible vers la classe du coté N.
Toutefois, et en particulier dans le cas d'une association N vers N, il est fréquent de promouvoir l'association au rang d'une classe pour augmenter la lisibilité ou en raison de la présence de l'association vers d'autres classes.

Exemple d'attribut - lien



L'association entre la classe *Etudiant* et la classe *Travail* est de type *N vers N*. La classe *Travail* décrit le sujet (l'énoncé) et la solution par l'étudiant n'est pas conservée (car pour un travail, plusieurs solutions existent mais peut contenir la solution unique).

Dans le cas des contrôle des connaissances, chaque étudiant compose individuellement sur un travail donné et la note obtenue ne peut être stockée ni dans un Etudiant en particulier (car il effectue de nombreux travaux dans sa carrière) ni dans un Travail donné (car il y a autant de notes que d'étudiant comme par exemple la liste des notes de test pour un ensemble d'étudiants, le test d'informatique pour les 1ère année).

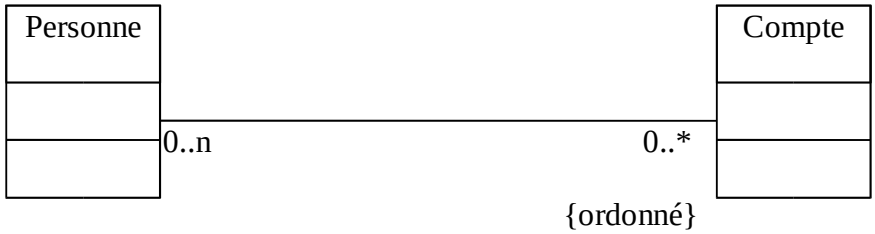
La classe **Résultat** ci-dessus est appelée une **classe-association**. Cette classe enferme l'attribut *note* de l'association entre Etudiant et Travail (on peut y mettre la solution de l'étudiant).

Un attribut de lien (classe - association) est une propriété des liens d'une association. Il correspond au schéma entité – relation dans les bases de données.

Remarque : un attribut – lien peut aussi apparaître pour une association 1 à 1 .

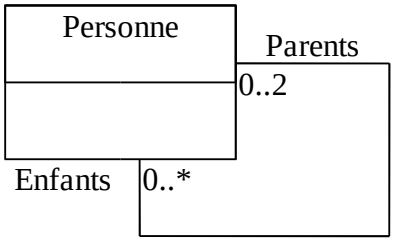
Contraintes sur les relations

Des contraintes peuvent être définies sur une relation. La multiplicité est un exemple de contraintes sur le nombre de liens qui peuvent exister entre deux objets. On représente une contrainte dans le diagramme entre accolades :



Ici, *{ordonné}* indique que la collection des comptes d'une personne est ordonnée.

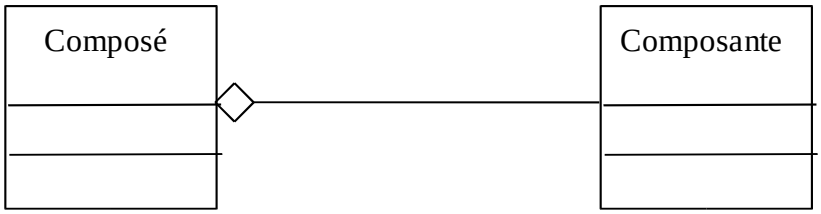
Les associations peuvent relier une classe à elle-même comme dans le cas des structures récursives. Ce type d'association est appelé association **réflexive**. Le nommage des rôles prend toute son importance pour distinguer les instances qui participent à la relation. L'exemple suivant montre la classe des personnes et la relation qui unie les parents et leurs enfants en vie.



Ici, toute personne possède de zéro à deux parents et de zéro à plusieurs enfants. Le nommage des rôles est essentiel à la clarté du diagramme.

Agrégation : structure Composé - Composante

Une agrégation représente une relation non symétriques dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. En général, l'agrégation concerne un seul rôle d'une association (côté composé).



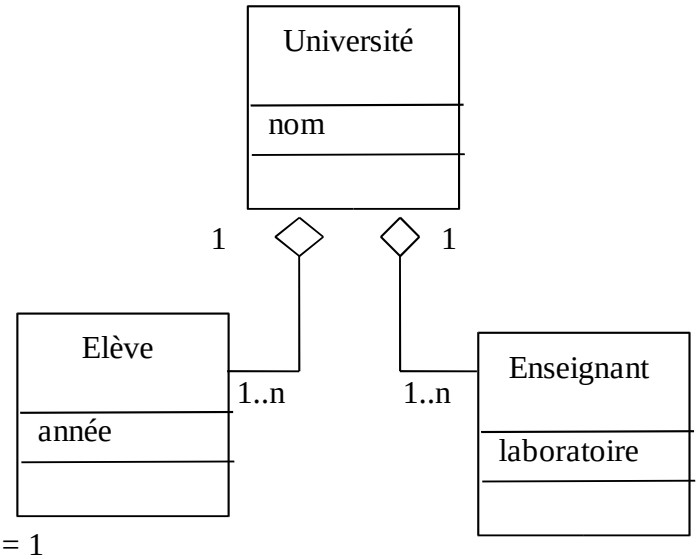
Dans une structure composé - composante, un petit losange apparaît du coté du composé. Le segment est dessiné depuis le composant vers le composé et le losange se trouve du coté du composé.

Les **critères** qui impliquent une agrégation sont :

- une classe fait partie d'une autre classe
- les valeurs d'attributs d'une classe se propagent dans les valeurs d'attributs d'une autre classe.
- une action sur une classe implique une action sur une autre classe (e.g. l'exemple table/pieds)
- les objets d'une classe sont subordonnés aux objets d'une autre classe.

L'inverse n'est pas toujours vrai : l'agrégation n'implique pas nécessairement les critères ci-dessus. Dans le doute, les associations sont préférables.

Exemple d'agréation



La multiplicité par défaut = 1
Une agrégation définit la relation "***partie-de***".

La multiplicité du coté de l'agrégat (composé) peut être supérieure à 1. Par exemple :

Ce diagramme montre que des personnes peuvent être copropriétaires des mêmes immeubles.

Différentes notions d'agrégation

Différentes notions :

Assemblage - Parties

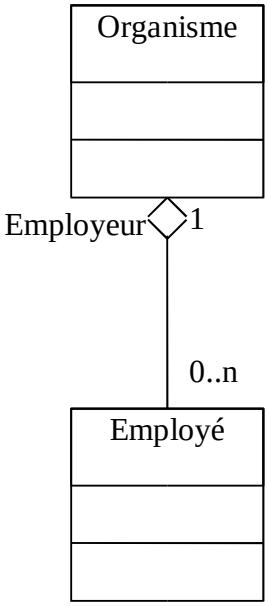
Collection - Membres
Contenant - Contenu

Exemple de d'agrégation-- *Assemblage-Parties*:

Un avion est composé de 1 à 4 moteurs. Un moteur appartient à 0 ou un avion.

Exemple d'agrégation -- *Contenant - Contenu* (on suppose que pour un vol donné, les pilotes sont à l'intérieur) **:**

Exemple de d'agrégation - *Collection-Membres* :



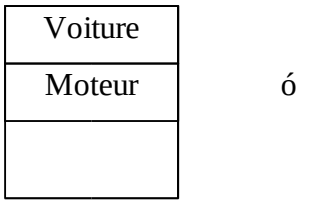
Remarque : la notion d'agrégation ne suppose aucune forme de réalisation particulière. La **contenance physique** est un cas particulier de l'agrégation, appelé **composition** tel que la disparition du contenant fait disparaître l'ensemble.

La composition (l'agrégation par valeur)

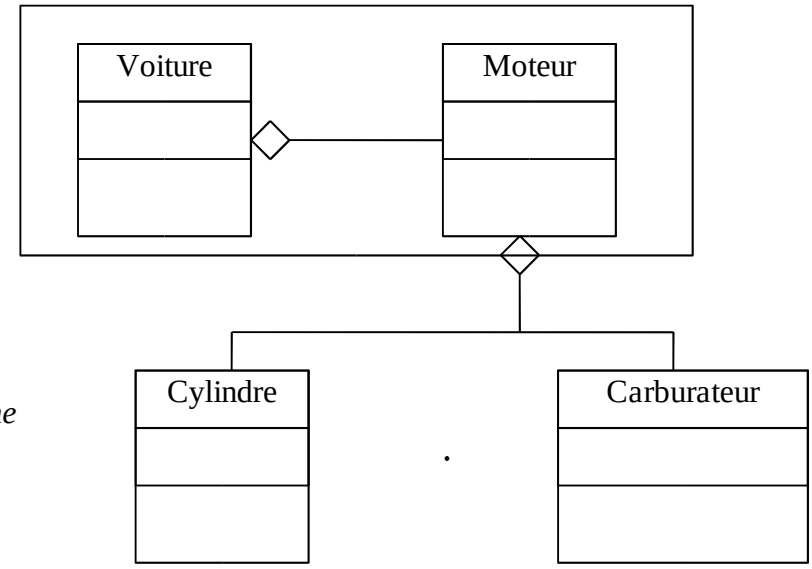
Les attributs constituent un cas particulier d'agrégation réalisée par valeur : ils sont physiquement contenus par l'agrégat. Cette forme d'agrégation est appelée composition.
NB : en UML, la composition (agrégation par valeur) se présente dans les diagramme par un losange de couleur noire. Pour simplifier les diagrammes, nous omettrons ce losange noir.

La composition implique une contrainte sur la valeur de multiplicité du coté de l'agrégat : elle ne peut prendre que les valeurs 0 ou 1.
La composition et les attributs sont interchangeables (sémantiquement équivalents). On utilise une composition à la place d'attribut quand cet attribut participe à d'autres relations dans le modèle.

Exemple :



ó



- *besoin de décrire le moteur*
- *le Moteur peut être lié à une autre classe*

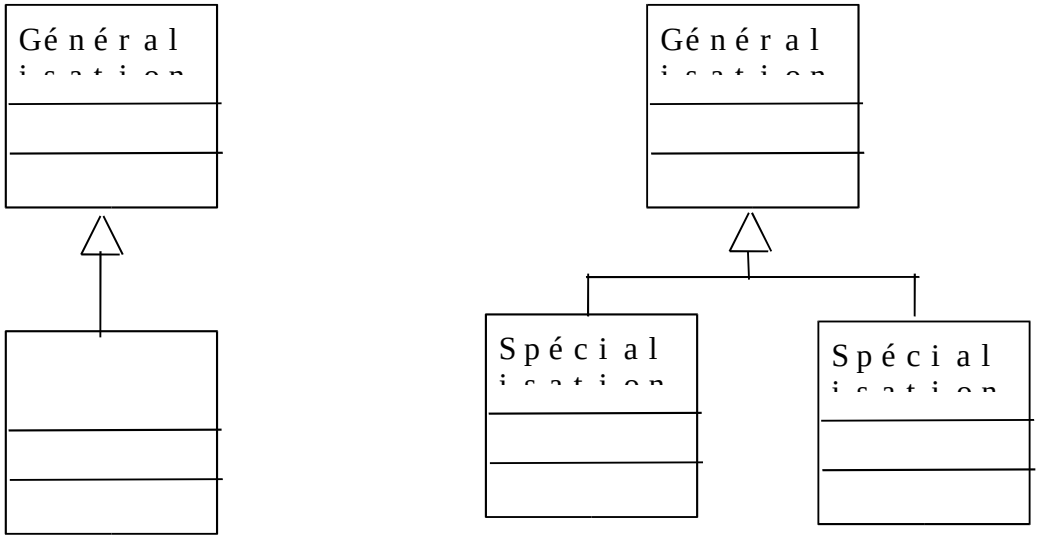
Dans le diagramme de gauche, *Moteur* est un attribut de *Voiture*. Dans le diagramme de droite, on établit une composition entre *Voiture* et *Moteur* car la classe *Moteur* participe à d'autres relations.

La navigation

Par défaut, une association (ou une agrégation) est navigable dans les deux sens. Dans certains cas, seule une seule direction de navigation est utile.

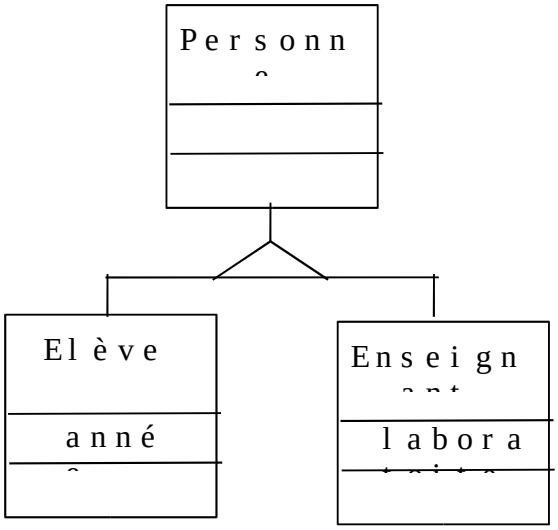
Dans l'exemple suivant, les objets instances de A voient les objets instances de B mais les objets instances de B ne voient pas les objets instances de A(e.g. banque – compte).

Structure Gén - Spéc (Héritage)



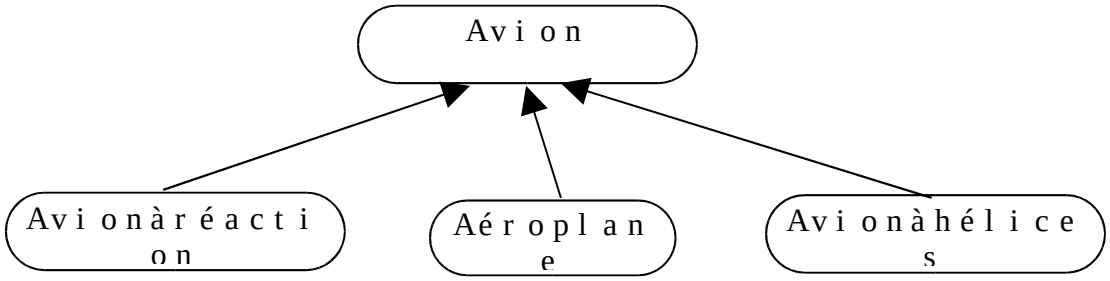
NB : La Généralisation - Spécialisation (Gen-Spec) définit la relation "**sorte-de**"

Exemple :

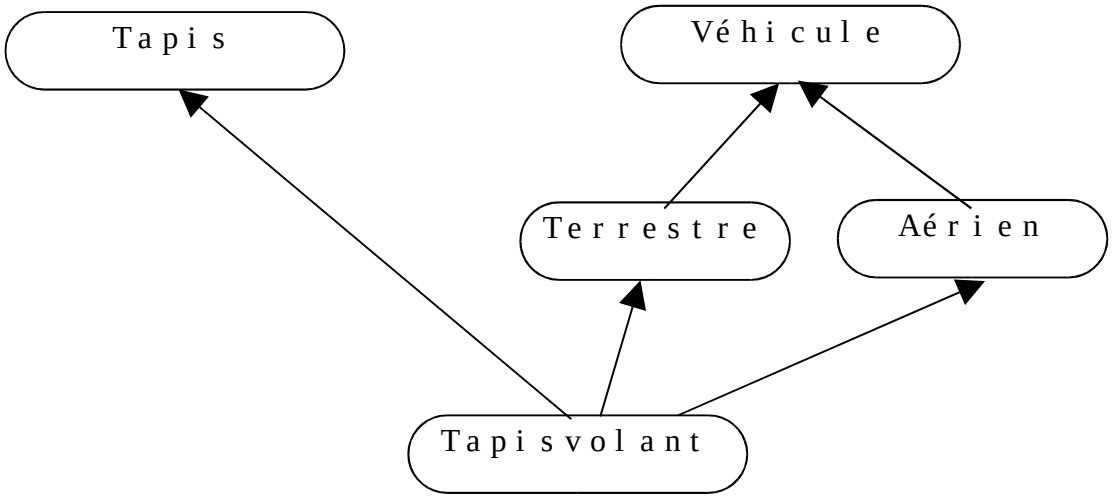


La classe Personne est appelée **super classe** et les classes Elève et Enseignant sont des **sous classes**.

Un autre exemple de hiérarchie Gén-Spéc :

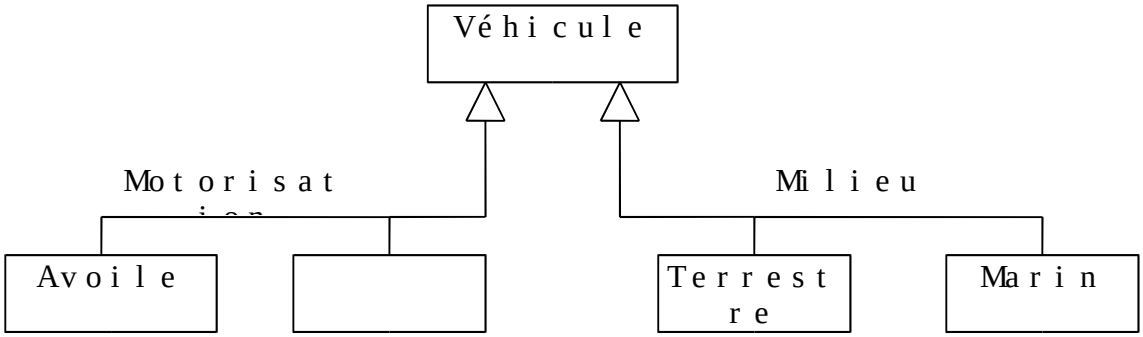


Exemple d'héritage multiple :



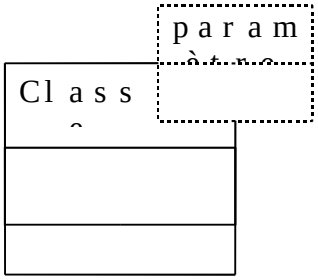
Une classe peut être spécialisée selon plusieurs critères simultanément. Dans ce cas, chaque critère est indiqué dans le diagramme :

Exemple :



Classes paramétrables

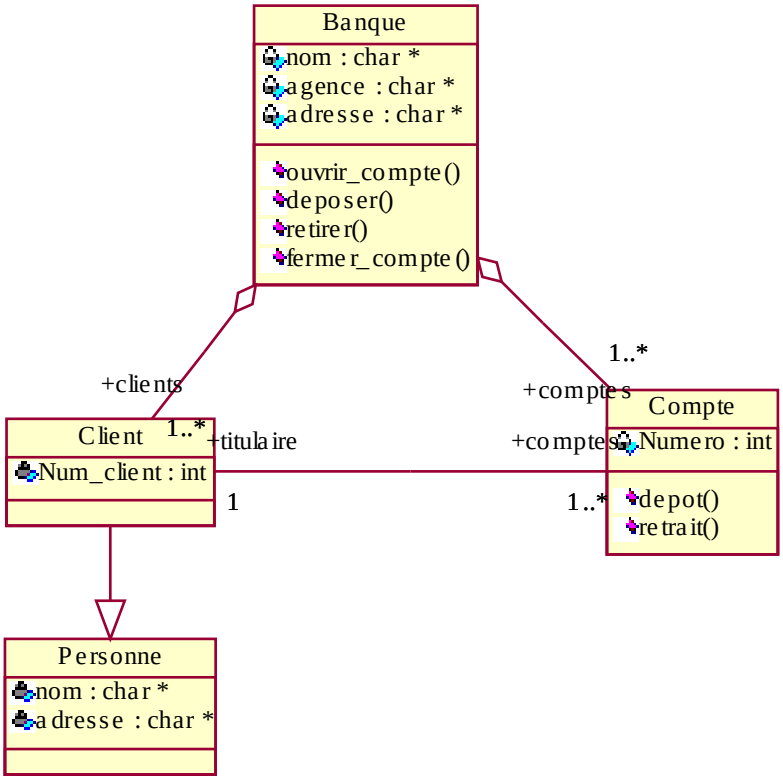
Dans le cas des classes génériques, une classe peut avoir un (ou plusieurs) paramètre :
Une classe paramétrable se présente sous la forme de :



Un exemple de classe paramétrable est un vecteur (ou liste, etc.) de "un type quelconque".

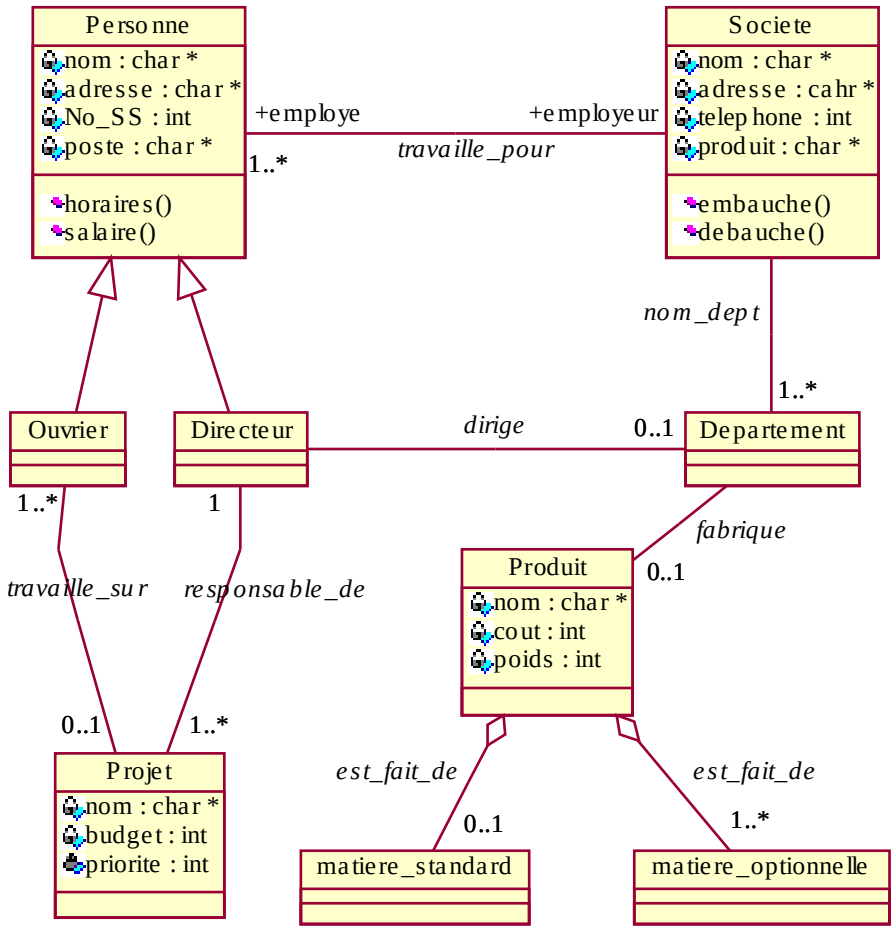
Un exemple : Banque

Modélisation d'une banque, ses clients et ses comptes. Un client est une sorte de Personne et peut avoir plusieurs comptes.



Un autre exemple : Société

Une entreprise emploie des Personnes. Elle est divisée en Départements et chaque Département est responsable de fabrication de 0 ou 1 Produit. Les Produits sont fabriqués par des matières Standards et/ou matières optionnelles. Les employés sont divisés En ouvrier ou en Directeur. Un ouvrier participe éventuellement à un Projet. Un Projet est dirigé par un Directeur.



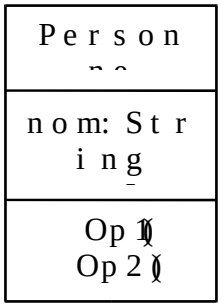
Génération de code

Des outils de spécification tels que "Rational Rose" ou "wclass" permettent de spécifier une application et de générer automatiquement le code pour ces spécifications.
Nous allons étudier le code généré pour les diagrammes de spécification vus ci-dessus. A chaque classe dU diagramme correspond une séquence de code générée. La combinaison de ces séquences donnera une application. L'utilisateur doit néanmoins compléter les fichiers générés, en particulier pour les fonctions membres qu'il définit.

Les outils de génération de code automatiques génèrent, pour une classe X, l'ensemble des constructeurs, destructeur, copie - constructeur et certains opérateurs (=, ==, !=, ..). Les fonctions membres déclarées sont également prises en compte.
Le code généré tient également compte des associations, agrégations, ...

Ci-dessous, nous examinons le code habituellement généré par les éditeurs de spécification objets. La classe *Personne* utilise la classe prédéfinie String (voir cours).

Classes

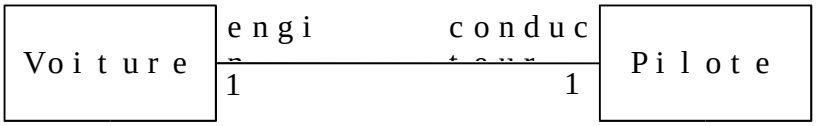



```
// Le fichier Personne.hpp généré automatiquement
class Personne
{ String nom;
int age;
public:
  Personne();                    // constructeur

Personne(const Personne& right);  // copie constructeur
  ~Personne();                   // destructeur

const Personne& operator=(const Personne& right);

int operator==(const Personne& right) const;

int operator!=(const Personne& right) const;
```

Remarque : Le générateur de code produit également un fichier "Personne.cpp" qui contient le code des fonctions déclarées dans "Personne.hpp". Dans certains générateurs de codes, les corps de ces fonctions sont laissés vides (l'utilisateur doit les compléter) alors que d'autres proposent des fonctions complètes.

Association 1 vers 1

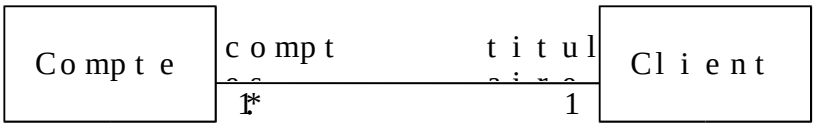
Une association 1 vers 1 est réalisée à l'aide des pointeurs placés dans les parties privées des classes
qui participent à l'association.
Dans l'exemple ci-dessous, "engin" est le rôle du coté voiture et "conducteur" est le rôle du coté Pilote. La multiplicité de l'association est 1 vers 1.




```
class Voiture
{
 public :
        const Pilote * get_conducteur() const;
        void set_conducteur(Pilote * value);
 private :
        Pilote * conducteur;
};
```

```
class Pilote
{
 public :
        const Voiture * get_engin() const;
        void set_engin(Voiture * value);
 private :
        Voiture * engin;
};
```

Association 1 vers N (ou N vers 1)



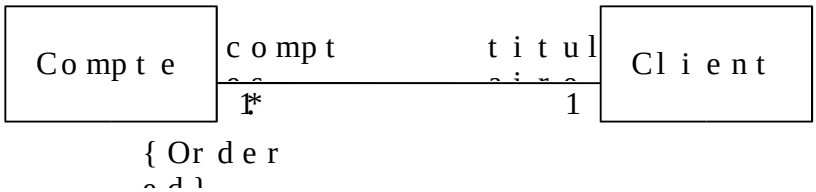
Le générateur de code réalise l'association par des pointeurs placés dans les parties privées des classes qui participent à l'association. La multiplicité 1..* (ou 1..*) est réalisée par un ensemble de taille non contrainte de pointeurs (type *Ensemble_Infini_de_pointeurs<X>* : un ensemble de taille non limitée de pointeurs sur X où X est un type). Le type ensemble est un type non ordonné.

```
class Client
{ ......
 private :
        Ensemble_Infini_de_Pointeurs<Compte>  comptes;

 public :
        const Ensemble_Infini_de_Pointeurs<Compte> get_comptes () const;
        void set_comptes (Ensemble_Infini_de_Pointeurs<Compte> value);
        ....
};

class Compte
{ Client * titulaire;
 public :
        const Client * get_titulaire() const;
        void set_titulaire (Client * value);
        ....
};
```

Remarques :
- Dans le cas de la classe *Client* ci-dessus, certains générateurs de codes utilisent le type prédéfini *Liste* à la place d'ensemble lorsque le type *Ensemble* n'est pas disponible. Le type *Liste* est cependant utilisé pour une collection ordonnée de taille illimitée.
- Grâce à la possibilité de création de types génériques (ou template) en C++ (voir plus loin), le types Ensemble ou Liste utilisé ci-dessus est en fait un type (classe) paramétré. Une classe paramétrée est un modèle de classes. Pour cela, on définit par exemple une classe "Liste de X" (notée Liste<X>) qui implante les opérations habituelles sur les listes sans connaître le type X de ses éléments. On instancie ensuite X par un type, par exemple le type entier (noté Liste<int>) et on obtient une liste d'entiers. Ainsi, en précisant un type pour X, on peut disposer d'une liste contenant n'importe quelle autre classe (par exemple, Liste<Personne>, Liste<Compte>, Ensemble<Voiture>, ...)

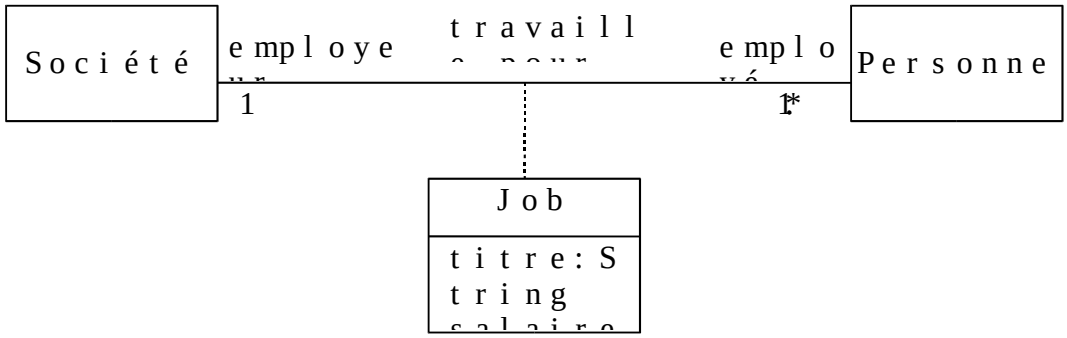
Pour les templates en C++, voir le cours plus loin.

Association 1 vers N avec une contrainte



Dans cet exemple, on impose la contrainte {Ordered} à l'association au niveau du rôle "comptes". Le code généré est tout à fait semblable à l'exemple précédent. La seule différence est que la classe Client doit contenir une collection ordonnée de "Compte". Ceci est implanté par l'utilisation d'une Liste à la place de l'ensemble dans la classe Client : *Liste_Infini_de_pointeurs<Compte>*.

Classe - association



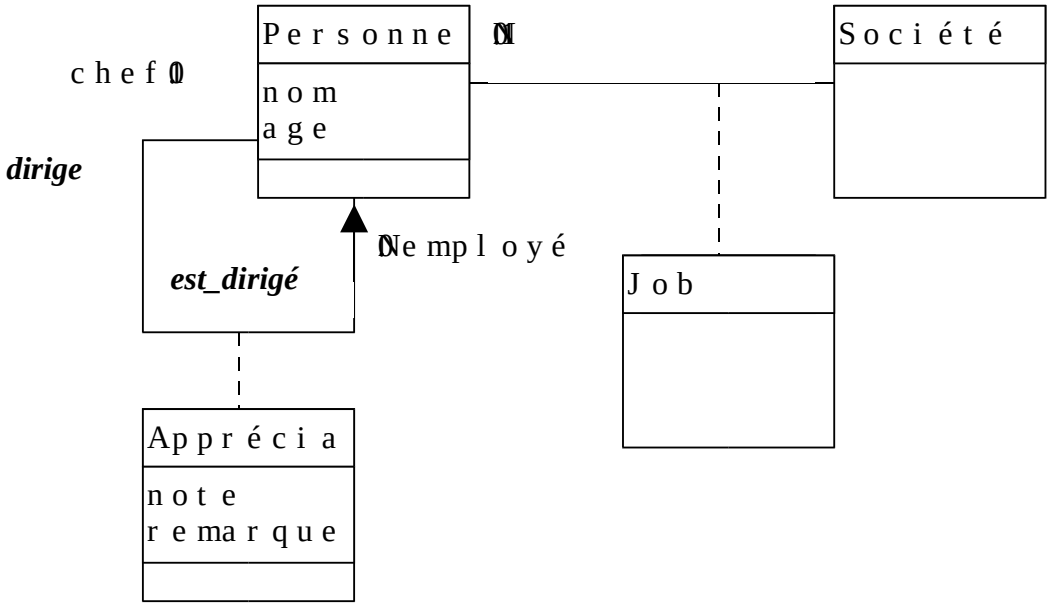

```
class Job
{private:
      String titre;
      int salaire;
      Personne *employé;
      Société *employeur;
 public :
      // fonctions get & set employé
      // fonctions get & set employeur
      // fonctions get & set titre
      // fonctions get & set salaire
};
```

```
class Société
{private :
      UnboundedSetByReference<Job> travaille_pour;
 public :
      const UnboundedSetByReference<Job> get_travaille_pour() const;
      void set_travaille_pour (UnboundedSetByReference<Job> value);
};
```

```
class Personne
{private :
      Job * travaille_pour;
 public :
      const Job * get_travaille_pour() const;
      void set_travaille_pour(Job * value);
};
```

On remarque que dans le cas d'une classe - association, la relation entre les classes Société et Personne devient indirecte et passe par la classe Job.

Un autre exemple d'attribut – lien



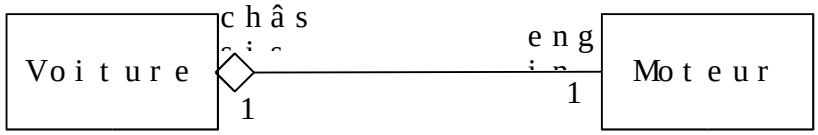

```
class Appreciation {                          class Personne {
      Personne * chef;                              unbound_set_ref<Personne> employé;
      Personne * employé;                           unbound_set_ref<Appreciation>
dirige;
      int note;                                     Appreciation * est_dirigé;
      String remarque;                                    String nom; int age;    ……
                                              ….
};                                            };
```

Remarques : il n'y aura plus de lien (i.e. instance de relation) directe entre Personne et Société; ce lien passera par Appréciation. C'est un exemple de relation (classe = abstraction) et son instance (lien = instance de la relation) proche des réalisation des schémas relationnels.
Pour comprendre le code, prendre les classe P (Personne), P'(Personne') avec un attribut - lien A (Appréciation) puis confondre P et P'.

Agrégation

Plusieurs types d'agrégation selon la multiplicité.

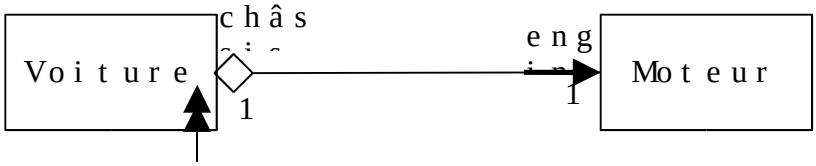
Agrégation 1 vers 1




```
class Voiture                          class Moteur
{Moteur * engin;                       {Voiture * chassis;
 public :                               public :
      const Moteur * get_engin() const;      const Voiture * get_chassis();
      void set_engin(Moteur * value);        void set_chassis(Voiture * value);
      ....                                   ....
};                                     };
```

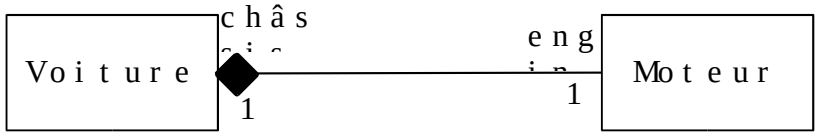
Agrégation à navigabilité restreinte

Si l'agrégation est à navigabilité restreinte comme dans l'exemple :



dans ce cas, la classe Moteur ne contiendra pas l'attribut Voiture et les fonctions get & set de cet attribut disparaissent.

Agrégation par valeur (composition)

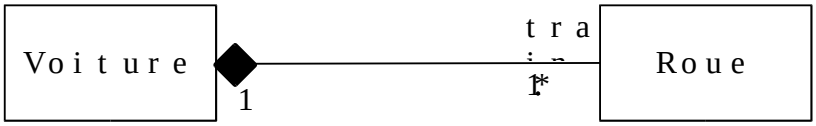


La seule différence est que dans la classe agrégat (ici Voiture), l'attribut Moteur (le composant) est représenté par valeur au lieu de référence. Il n'y aura pas de changement dans la classe composant (ici Moteur).

```
class Voiture                          class Moteur
{Moteur  engin;                        {Voiture * chassis;
 public :                               public :
      const Moteur  get_engin() const;       const Voiture * get_chassis();
      void set_engin(const Moteur  value);   void set_chassis(Voiture * value);
      ....                                   ....
};                                     };
```

N.B. la composition doit être vue comme la contenance : l'agrégat contient les composantes de sorte que si les composantes disparaissent l'ensemble (l'agrégat) disparaît, e.g. Voiture et Moteur.

Agrégation (composition) par valeur 1 vers N



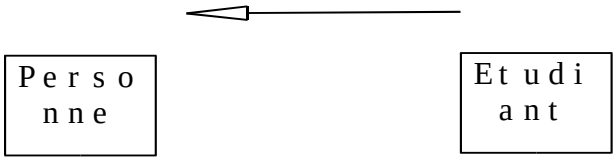
Il n'y aura pas de changement dans la classe composant (Roue). Par contre, la classe composée (agrégat) change et on utilise un ensemble infini par valeur de composants :

```
class Voiture
{Ensemble_Infini_par_Valeur<Roue>  train;
public :
        const Ensemble_Infini_par_Valeur<Roue> get_train() const;
        void set_train(const Ensemble_Infini_par_Valeur<Roue> value);
        ....
};
```

Le type Ensemble_Infini_par_Valeur<Roue> est un type générique (template) et représente un ensemble non ordonné infini de Roue (par opposition à un ensemble de pointeurs sur Roue).

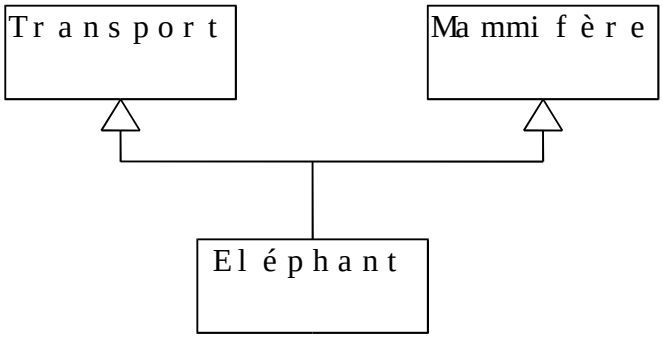
Héritage

On peut avoir deux types d'héritages : simple et multiple.



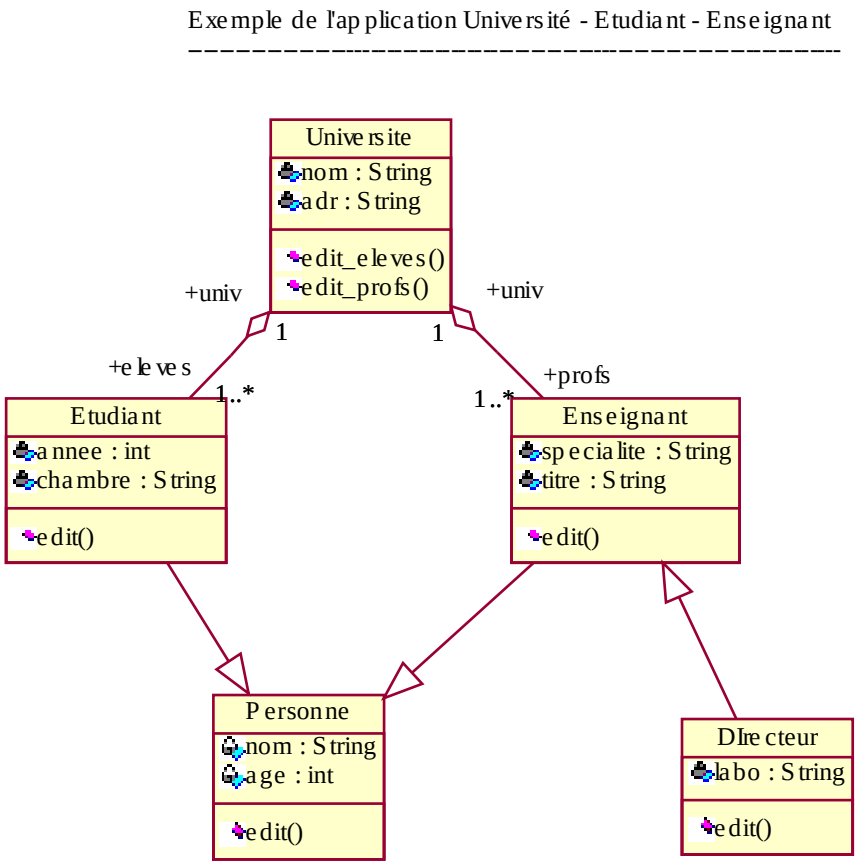

```
class Etudiant : public Personne
{
        ..........
};
```

Un autre exemple :




```
class Eléphant : public Transport, public Mammifère
{
        .....
};
```

Exemple


Exemple de l'application Université - Etudiant - Enseignant
Universite
nom : String
adr : String
edit_eleves()
edit_profs()
+univ
1
+univ
1
+eleves
1..*
+profs
1..*
Etudiant
annee : int
chambre : String
edit()
Enseignant
specialite : String
titre : String
edit()
Personne
nom : String
age : int
edit()
DIrecteur
labo : String
edit()

Templates (fonctions et classes génériques)

Les *Templates* permettent de définir des fonctions capables d'effectuer des opérations sur des types de données qui seront définis au moment de l'utilisation.
Par exemple, on peut envisager la fonction *max* capable de calculer le maximum de deux valeurs V1 et V2 de type T :

```
T max(T v1, T v2)
{if (v1 > v2) return v1 ;
 else retuen v2;
}
```

Pour que cette fonction soit utilisable, il faut que l'opérateur '>' soit défini sur le type T. La fonction *max* pourra alors admettre n'importe quel couple de paramètres et calculer leur maximum.

De même, on peut envisager une fonction de tri de vecteur de données qui accepte des vecteurs de différents types.

Habituellement (particulièrement en C), le problème de définir une fonction qui réalise un calcul sur des types de données différentes est résolu par :
1- définition de plusieurs fonctions ou la surcharge de fonction. L'inconvénient dans ce cas est la gestion et la maintenance de ces fonctions.

2- définition d'une fonction avec des arguments pointeurs génériques. L'inconvénient est l'absence de contrôle de types et la conversion des pointeurs est à la charge de l'utilisateur.

D'autres solutions comportant des inconvénients peuvent être envisagées.

La solution proposée par les Templates en C++ permet de définir un modèle de fonctions.

Fonctions génériques

Exemple d'une fonction générique en C++

```
template <class T>
T max(T x, T y)   {return (x > y ? x : y ; }

class n_importe
{       .....
        bool operator>(const n_importe t1, const n_importe t2) { .... }    // opérateur "plus grand"
};

void main()
{n_importe a, b, c;
 int x, y, z;
 a = max(b, c);          // appel de la fonction "max" avec des paramètres de type "n_importe"
 x = max(y, z);          //          ==                    ==          "int"
}
```

Dans cet exemple, la fonction *max* accepte deux paramètres de n'importe quel type et calcule leur maximum. Pour les objets de la classe "n_importe", la fonction de comparaison '>' est utilisée.
La présence de la définition de l'opérateur '>' pour le types utilisés est importante. La classe "n_importe" ci-dessus le définit et le type prédéfini de base "int" dispose de cet opérateur.

La fonction générique "max" accepte de calculer le maximum de n'importe quel paires d'objets (du même type T). Pour cela, elle utilise la comparaison '>'. On notera que si la comparaison d'objets des types de base (int, float, double, char) est simple, il n'en n'est pas de même pour les autres types (par exemple char*, voiture, individus, table, chaise, etc.).

L'intérêt de la fonction générique *max* est évident. Il n'est pas réaliste de connaître tous les types qui peuvent figurer en paramètre de "max" et de proposer une version de "max" pour ces types.

Le type T est un type formel, on peut remplacer T par n'importe quel autre nom mais le même nom doit apparaître aux mêmes endroits que T. T est appelé le paramètre générique. Le type effectif de T sera précisé lors de l'utilisation de la fonction *max* comme dans la fonction *main* ci-dessus.

On peut avoir plusieurs paramètre générique :

```
template <class U, class V>
int f(U a, V b, ...) { ... }
```

En plus de la définition générique, on peut évidemment continuer à utiliser le nom *max* pour définir d'autres fonctions ou d'en surcharger d'autres.
Par exemple, en plus de la définition de *max* ci-dessus, on peut avoir la définition explicite :

```
char * max(char * x, char * y)
        {return (strcmp(x, y) > 0) ? x ; y; }
```

Lors de l'appel de la fonction "max", il y a d'abord une recherche parmi les définitions explicites et/ ou surchargées (les définitions non génériques). La recherche d'une fonction générique est entamée si les définitions explicites ou surchargées ne conviennent pas.

Classes génériques

Un utilisateur peut avoir besoin de manipuler des structures de données s'appuyant sur des types différents : par exemple, il aura besoin d'une pile d'entiers, de réels, de char, ...

Les classes génériques permettent de réaliser des structures complexes manipulant des types de bases différents.

Exemple de la classe Pile générique :

```
template <class T> class Pile              // classe Pile générique
{ T * contenu;
  int taille;
  int top;
  public :
        Pile(int t) : taille(t), top(-1) {contenu = new T[t]; }
        ~Pile() {delete [] contenu;}
        void empiler(const T &e) {contenu[++top] = e;}
        T & depiler() {return contenu[top—];}
};

void main()
{Pile<int> Pi(5); Pi.empiler(3); ..          // définition d'une pile de 5 entiers
 Pile<char> Pc(10); Pc.empiler('x'); .... // définition d'une pile de 10 caractères
}
```

Remarque :

Il est possible d'alléger les notations à l'aide de *typedef* :

```
typedef Pile<double> Pile_dble;

Pile_dble Pd(10);
Pd.empiler(3.5);
.....
```

De même, à l'intérieure d'une classe générique :

```
template <class T> class thing
{...
 public :
 typedef T T;              // définit thing<T>::T
};
```

Les méthodes non inline d'une classe générique utilisent une syntaxe assez lourde. Par exemple, pour définir la fonction *empiler* de la classe Pile générique ci-dessus en dehors de la classe :

```
template <class T> void Pile<T>::empiler(const T& e)          {...}
```

Une classe générique peut être paramétrée par des types et par des variables. Par exemple, on peut définir la classe Pile générique en précisant un paramètre qui fixe la taille de la pile :

```
template <class T, int length> class Pile              // length sera la taille de la pile
{ T * contenu;
  int taille;
  int top;
  public :
      Pile() : taille(length), top(-1) {contenu = new T[t]; }
      ~Pile() {delete [] contenu;}
      void empiler(const T &e) {contenu[++top] = e;}
      T & depiler() {return contenu[top—];}
};

void main()
{Pile<int, 5> Pi; Pi.empiler(3); ..          // définition d'une pile de 5 entiers
 Pile<char,10> Pc; Pc.empiler('x'); ....  // définition d'une pile de 10 caractères
}
```

Dans certains cas, une classe générique peut ne pas convenir à certains types. Par exemple, on aurait certaines difficultés dans le cas d'une pile de char * (problème d'affectation de chaînes, problème de comparaison de chaînes, ...).

Dans ce cas, une solution consiste à définir une instance spécifique de la classe Pile :

```
class Pile<char *>                  // définition spécifique de Pile de chaînes
{
      .....
      // les mêmes opérations mais en utilisant strcpy, strcmp, ..... pour les chaînes
};
```

Dans ce cas, lors d'une instanciation, les instance spécifiques sont d'abord prises en compte avant d'effectuer des instanciations automatiques :

```
void main()
{Pile <char *> Pch(15);      // une pile de 15 chaînes de caractères
 Pch.empiler("Galata"); ...
}
```

Ici, l'instance *Pch* est définie à l'aide de la définition spécifique de la Pile pour char*.

Une autre solution à ce problème consiste à traiter le cas particulier de l'affectation. En effet, on constate que la seule différence entre la classe Pile générique ci-dessus et la version spécifique pour char* est dans la fonction *empiler* et plus particulièrement dans l'affectation *contenu[++top] = e*.
On peut donc définir une fonction générique appelée *affect* et remplacer l'affectation ci-dessus par un appel à *affect*. Pour traiter le cas de *char**, on définira une version spécifique de la fonction *affect* pour *char**. L'exemple de la page suivante contient la fonction *affet*.

Remarque : le langage C (et C++) effectue une conversion implicite lors de l'appel d'une fonction. Par exemple, avec :

```
void f(int a, int b) { ....}
```

et l'appel :

```
f(2.5, 6.7);
```

il y a une conversion des réels en entiers avant l'appel de la fonction f (un warning possible)
Par contre, dans le cas des fonctions génériques, il n'y a pas de conversion implicite.

Exemple-1 :
Considérons la fonction générique *max* vue précédemment et l'exemple d'utilisation suivante :

```
int A=1; char B='a'; int C=max(A, B);          // problème de conversion
```

Cet appel à *max* pose problème car il n'y aura pas de conversion implicite de *char* en *int*.
Il faut appliquer une conversion explicite sur la variable B :

```
int A=1; char B='a'; int C=max(A, (int) B);    // conversion explicite
```

Exemple-2 :
soit la fonction générique sqrt :

```
template <class T> T sqrt(T) {...}
```

et la fonction *f* ci-dessous déclare :

```
void f(int i, double d, complexe c)
{complexe z1 = sqrt(i);              // sqrt(int)
 complexe z1 = sqrt(d);              // sqrt(double)
 complexe z1 = sqrt(c);              // sqrt(complexe)
}
```

Si l'utilisateur fait par exemple un appel à *sqrt(double)* avec un argument *int*, une conversion explicite doit être utilisée :

```
void f(int i, double d, complexe c)
{complexe z1 = sqrt((double) i);     // sqrt(double)
 complexe z1 = sqrt(d);              // sqrt(double)
 complexe z1 = sqrt(c);              // sqrt(complexe)
}
```

Un exemple

Exemple classe générique Vecteur

```
template <class T> class Vecteur
{int nb_elements,max_elements;
  T * contenu;              // contiendra les éléments du vecteur
 public :
       Vecteur(int n=0) : nb_elements(0), max_elements(n)
              {if (n==0) contenu=NULL; else contenu=new T[n];}
       void ajouter(const T& e)
              {if (nb_elements < max_elements)
                     affect(contenu[nb_elements++],e);
              }
       int size() {return nb_elements;}
       T& operator[](int i)          // renvoyer le ième élément
              {assert (i>=0 && i<nb_elements);
               return contenu[i];
              }
       friend ostream& operator<<(ostream & f, Vecteur<T> & V)
       {f << '<'<<V.nb_elements<<'>';
        for (int i=0; i< V.nb_elements ; i++) f << V.contenu[i] << ", ";
        return f<<endl;
       }
};
```

La fonction **affect** permet une simple affectation dans le cas des types et classes ayant défini l'affectation (opérateur =). Par contre, pour les chaînes de caractères, affect utilisera *strcpy* avec une allocation pour l'opérande gauche.

```
template <class T> void affect(T& g, const T& d)   // pour les types ayant l'opérateur '='
       {g=d;}
```

Pour le type char * :

```
void affect(char * &g, const char * d)                  // pour char *
       {g=new char[strlen(d)+1]; strcpy(g,d);}
```

Pour les besoins de comparaison des éléments, on définit :

```
template <class T> class Comparacteur
{public :
       static int plus_petit( T& a,  T& b)  {return (a<b);}          // pour static, voir plus loin
};
```

Et pour permettre la comparaison des chaînes de caractères, on définira la version spécifique :

```
class Comparacteur<char *>
{public :
       static int plus_petit(const char* a, const char* b)
              {return (strcmp(a,b) < 0); }
};
```

Et enfin la classe des vecteurs triables (on peut les trier dans l'ordre croissant) est dérivée de la classe Vecteur et de la classe Comparateur. L'héritage de Comparateur permet de disposer de l'opération "plus_petit" :

```
template <class T> class Vecteur_triable :          // héritage double
        public Vecteur<T>, public Comparacteur<T>
{public :
    Vecteur_triable(int s) : Vecteur<T>(s)          // constructeur
        {}
};
```

On peut définir la fonction générique de tri des vecteurs par (méthode Bulle) :

```
template <class T> void tri(Vecteur_triable<T> & V)
{int n=V.size();
  for (int i=0; i< n-1; i++)                        // tri par la méthode bulle
    for (int j=n-1; i<j; j--)
        if (V.plus_petit(V[j], V[j-1]))             // échanger les 2 éléments
            echange(V[j], V[j-1]);
}
```

La fonction échange permet de permuter deux éléments existants. Le fait que ces éléments existent nous empêche d'utiliser la fonction affect ci-dessus car affect, dans sa version des char* alloue une zone de mémoire pour son opérande gauche supposé inexistant.

```
template <class T> void echange(T& g,  T& d)        // échange générique
    {T temp=g; g=d;d=temp;}
```

Pour char * :

```
void echange(char * &g, char *& d)                  // échange de deux chaînes
{char * temp=new char[strlen(d)+1]; strcpy(temp,g);  // ou affect(temp, g);
  strcpy(g,d); strcpy(d,temp);                          // Attention aux tailles variables
}
```

Un exemple d'utilisation : vecteur d'entiers, vecteur de char *, vecteur de String, ...
Rappel : ici, seul le type char * nécessite des versions spécifiques d'affectation (fonction affect), d'échange (fonction echange) et de la classe spécifique Comparateur (pour l'opérateur '<').
Les autres classes (y compris les classes de bases) disposent de ces opérateurs.

```
void main()
{Vecteur_triable<int> Vi(5);
 Vi.ajouter(15);Vi.ajouter(25);Vi.ajouter(5); Vi.ajouter(2);Vi.ajouter(30); cout << Vi;
 tri(Vi);cout << Vi;

 Vecteur_triable<char *> Vch(5);
 Vch.ajouter("toto"); Vch.ajouter("tata"); Vch.ajouter("titi"); cout << Vch;
 tri(Vch); cout << Vch;

 Vecteur_triable<String> Vstr(5);
 Vstr.ajouter("blabla"); Vstr.ajouter("ratata");  Vstr.ajouter("blobert"); cout << Vstr;
 tri(Vstr); cout << Vstr;
}
```

Remarques :

Classes génériques et fonctions amies :
Une fonction amie (friend) qui fait référence à un paramètre générique doit elle aussi être générique.
Une fonction amie générique n'est amie que des instances de la classe générique qui ont les mêmes paramètres génériques.

Classe générique et attribut de classe :
Les attributs de classe déclarés avec le mot clé **static** sont communs à tous les objets instances d'une instance particulière d'une classe générique.

Dans l'exemple ci-dessus, on a déclaré static la fonction **plus_petit**. Elle sera commune à tous les objets instances d'une instance particulière, par exemple **Comparateur<int>** (de manière directe) ou de **Vecteur_triable<double>** (par héritage). En d'autres termes, tous les vecteurs triables de doubles partagent la même fonction.

Exemple : Vecteur générique

L'exemple ci-dessous définit un vecteur générique. Pour simplifier, nous ne traitons pas le cas de *char**. L'utilisateur pourra utiliser la classe String (défini en cours) à la place de char *.

Dans cette solution, les insertions ne sont pas ordonnées.

Classe vecteur générique

```
#include <iostream.h>
#include "String.hpp"

#define N 5                 // l'incrément N, 2N, 3N, ...
```

Remarque : N peut être paramètre générique : *template <class T, int N> class vecteur {…};*
ou peut être paramètre du constructeur du vecteur. (voir la fonction main)

```
template <class T>
 class vecteur
{int nb_elements;                    // nombre d'éléments dans vecteur
 int max_elements;              // nombre maximum possible d'éléments (N, 2N, 3N..)
 T* tab;
 public:
 vecteur()
        {max_elements=N; nb_elements=0;
         tab=new T[N];
        }

 vecteur(const vecteur<T>& V)
        {max_elements=V.max_elements;nb_elements=V.nb_elements;
         tab=new T[max_elements];
         for (int i=0; i<nb_elements; i++)        tab[i]=V.tab[i];
        }

 vecteur<T>& operator=(const vecteur<T>& V)
        {if (this == &V) return *this;
         this->~vecteur();
         max_elements=V.max_elements; nb_elements=V.nb_elements;
         tab=new T[V.max_elements];
         for (int i=0; i<nb_elements; i++)        tab[i]=V.tab[i];
         return *this;
        }

 int operator==(const vecteur<T>& V)
        {if (nb_elements != V.nb_elements) return 0;
         for (int i=0;  i<nb_elements; i++)
                if(tab[i] != V.tab[i]) return 0;
        return 1;
        }

 int operator!=(const vecteur<T>& V)
        {return !(*this == V);}
```

```
vecteur<T>& operator+=(const T& ch)
{if (nb_elements < max_elements)
      {tab[nb_elements]=ch;        nb_elements++; return *this;
      }
 else
      {int i; T* t1=new T[max_elements+N];
      for ( i=0; i<nb_elements; i++)     t1[i]=tab[i];
      this->~vecteur();
      t1[nb_elements]=ch;
      tab=t1;  nb_elements++; max_elements+=N;
      }
 return *this;
}

vecteur<T> operator+(const T& ch)
{vecteur<T> V=*this;   V += ch;   return V;}

const T& operator[](int i) const
{if ((i<0) ||(i>=nb_elements)) return NULL;
 return tab[i];
}

/* si T = int, il y aura une ambiguïté par rapport a l'opérateur [int]
int operator[](const T& ch) const              // renvoie l'indice de ch
{for (int i=0; i<nb_elements; i++)             // renvoie -1 si non trouve'
      if (tab[i] == ch) return i;
 return -1;
}
*/

vecteur<T> operator+(const vecteur<T> & V)          // FUSION
 {vecteur<T> V1=*this;
  for (int i=0; i<V.nb_elements; i++) V1 += V[i];       // ou V.tab[i]
  return V1;
  }

~vecteur()        {delete [] tab; }

friend ostream& operator<<(ostream& O, const vecteur<T> & V)
{O << "<" << V.max_elements <<", "<<V.nb_elements<<"> ";
  for (int i=0; i<V.nb_elements; i++) O<<V.tab[i] << " ";
  return O << endl;
}

friend istream& operator>>(istream& Is, vecteur<T> & V)
{cout << "donner " << V.max_elements << " elements, 1 par ligne \n";
 T ch;
 for (int i=0; i<V.max_elements; i++) {Is >> ch; V += ch;}
 return Is;
}
};
```

```
void main()
{vecteur<int> Vi1; Vi1 += 12; cout << Vi1;
 vecteur<String> V1;
 V1 += "abc";
 V1 += "def";
 V1 += "jkl";
 V1 += "mnp";
 V1 += "opq"; cout << V1;

 vecteur<String> V2=V1; cout << V2 ;
 vecteur<String> V3; V3=V1; cout << V3;
 cout << V1 + "xyz" ;
 vecteur<String> V4; V4 = V1 + "uvw" + "xyz" ;
 cout << V4;
 V1 = V1 + "123"; cout << V1;
 if (V1 == V4) cout << "egalite de V1 et V4";
 else cout << "difference de V1 et V4";
 cout << endl;
 cout << "le 3eme element de V1 est = " << V1[3] << endl;
 cout << "la fusion de V1 et V3 donne" << V1+V3 << endl;
 vecteur<String> V5; cin >> V5; cout << V5;
}
```

Remarques :

- Si N était paramètre générique, on pourrait déclarer :

vecteur<string, 25> Vs; …..

Cette solution est quelque peu rigide car elle fixe la taille du vecteur dès sa création.

- Si N était paramètre du constructeur de vecteur, on pourrait déclarer :

vecteur<string> Vs(25); …..

Cette solution est moins rigide car elle permet de déclarer un vecteur d'une certaine taille. Egalement, on pourra envisager un constructeur sans paramètre et faire évoluer le vecteur plus tard.

TP

TP classe générique:

1- Créer un répertoire avec une table générique. On pourra y stocker des couples (string x string) pour le nom et le n° de téléphone ou des couples (int , string) pour un n° et une adresse, etc.
2- Création d'une file d'attente d'objets génériques.
3- Créer une classe générique "container" commune à un ensemble de types tels que table, pile, file, set, map, etc.

TPs de la première partie du cours POO

Plusieurs donnés en exemple et traités dans ce texte.

Classes :

date (donné en exemple, demander à compléter)
temps (à demander)
string (donné en exemple dans ce texte)
vecteur (avec agrandissement) (donné en exemple dans ce texte)
complexes (donné en exemple dans ce texte)
liste (donné en exemple, a compléter)
pile (donné en exemple dans ce texte)
file (à demander)
graphe (à demander)
personne (donné en exemple, à utiliser)
.....

Exercices, TPs, Applications en POO

TP Dico (donné en TP)
TP trains (à donner en TP)
TP Banque (schéma UML donné, demander en TP)
TP Entreprise (héritage) (demander en TP)
TP Circuits (demander en TP)
TP Interface graphique (demander en TP)

Liens avec les TDA

La traduction des TDA en C++ (Java également) est soumise à une convention simple.
A expliquer.

Bibliothèque STL

Voir le site SGI.
En attendant, vous pouvez voir le site http://www.ensta.fr/~diam/c++/online/STL_doc/
pour une documentation en français de STL.