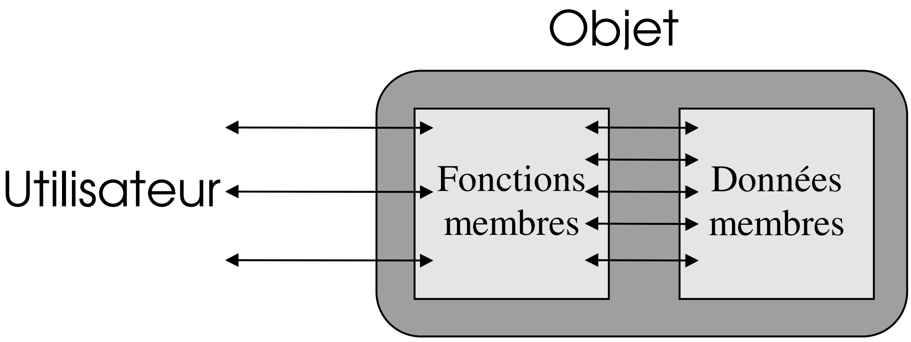
CLASSES.

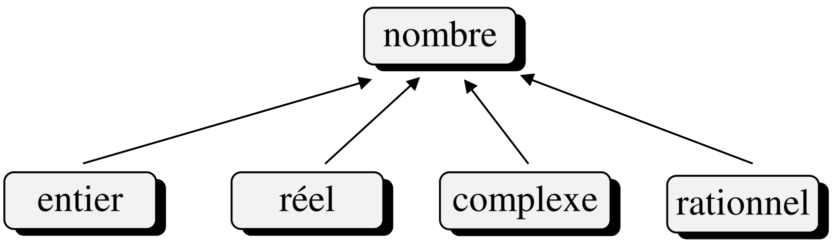
INTRODUCTION

Un langage orienté objet se doit de posséder les trois caractéristiques suivantes :

Encapsulation : {EX "encapsulation"**}**les données et les actions ou fonctions qui les manipulent sont regroupées sous une même entité : l'objet. Les actions sont alors appelées **méthodes** (ou fonctions membres en C++).Les détails de l'implémentation peuvent être cachés, l'utilisateur n'y ayant pas accès. L'utilisateur n'a accès qu'aux méthodes constituant l'interface de l'objet. Les données membres sont aussi appelées des **champs** ou **attributs**.**{**EX "fonction:membre"**}**



Héritage : {EX "héritage"**}{**EX "héritage:multiple"**}**l'héritage permet de définir de nouvelles classes à partir de classes préexistantes. Il permet d'utiliser ce qui existe déjà afin de le spécialiser ou de le généraliser. Quand l'héritage se fait à partir de plusieurs autres classes, on parle d'héritage multiple. C++ possède l'héritage multiple. L'héritage permet de définir des hiérarchies de classes. L'héritage est un concept essentiel pour le *réutilisabilité* ***{*** EX "*réutilisabilité*" ***}*** des classes.



Les flèches sont orientées des classes héritières vers les classes ancêtres.

Polymorphisme : {EX "polymorphisme"**}**Le polymorphisme consiste à donner un même nom à une action différente à chaque niveau de la hiérarchie : si le compilateur peut résoudre le problème de savoir de quelle fonction il s'agit, on parle de polymorphisme statique, en revanche si cette résolution est reportée au moment de l'exécution, et est fonction du type d'objet auquel est appliqué cette action, on parle de polymorphisme dynamique. La surdéfinition des fonctions est une forme d polymorphisme statique.

pourquoi masquer l'information ?

La façon dont sont représentées les données n'intéresse pas l'utilisateur de la classe : si j'utilise une instance de la classe *TDurée*, je ne suis pas concerné par le fait que la durée est représentée par 3 entiers non signés ou par un entier long non signé contenant sa conversion en secondes.

On se protège ainsi d'une mauvaise utilisation de la part de l'utilisateur de la classe : Si j'utilise un type *TDate* constitué de trois entiers jour, mois, année, le fait de ne pouvoir affecter les données membres que via des fonctions membres permettra de vérifier que le triplet (jour, mois, année) est une date légale.

SYNTAXE.

Une classe est composées de trois zones : **{**EX "private"**}{**EX "protected"**}{**EX "public"**}**

```
class X
{  private :        // partie accessible uniquement aux fonctions
                    // membres de la classe et aux fonctions amies
   protected :      // partie accessible aux membres et amies de la
                    // classe ainsi qu'aux classes dérivées
   public :         // accessible à tout utilisateur d'une instance
                    // de la classe
};
```

S'il n'existe pas d'en-tête (*private*, *protected*, ou *public*) il s'agit par défaut de la zone *private*.

Une déclaration d'une instance**{**EX "instance"**}** de la classe *X* se fait de la façon suivante :

```
X x;
```

Un appel d'une fonction *f* membre de la classe *X* se fait de la façon suivante :

```
x.f( .. )
```

exemple :

```
class TDuree
{ private :  unsigned int heure, minute, seconde;
             unsigned long conv();
             void invconv( unsigned long);
  public :   void ecrire ();
             TDuree operator + (TDuree d);
}
```

La fonction membre *conv* est définie de la façon suivante :

```
unsigned long TDuree::conv()
{   return  heure*3600L+minute*60+seconde;}
```

Les champs heure, minute, seconde sont les champs de l'instance pour laquelle est appelée la fonction conv. Devant chacun des champs est sous entendu ***(*this).***, qui s'écrit aussi **this->.** Pour chaque fonction membre, on peut utiliser le pointeur *this* qui pointe sur l'instance pour laquelle est appelée la fonction membre. L'opérateur **::** est l'opérateur de portée**{**EX "opérateur:portée"**}** et sert à indiquer au compilateur qu'on est en train de déclarer une fonction membre de la classe qui est opérande de gauche de l'opérateur.

```
void TDuree::invconv(unsigned long x)
{heure=x/3600;minute=(x%3600)/60;seconde=x%60;}
```

opérande de droite de l'opérateur
l'opérande de gauche est l'instance

```
TDuree TDuree::operator + (TDuree d)
{ return invconv( conv() + d.conv());}
```

CONSTRUCTEUR, DESTRUCTEUR, INITIALISATION DES DONNEES MEMBRES.

{EX "constructeur" \b**}{**EX "destructeur" \b**}**

constructeur

Un constructeur est une fonction membre sans type qui porte le même nom que la classe, et ne retourne pas de valeur. Un constructeur est appelé au moment de la création d'une instance de la classe : un constructeur sert à initialiser les données membres de la nouvelle instance :

- Les données membres publiques d'une instance de classe ne contenant que des données membres publiques, peuvent être initialisées avec des accolades comme les données membres des structures.
- En revanche les membres privés ne peuvent être initialisés que par un constructeur.

Une création d'instance se fait de façon explicite ou de façon implicite :

- explicite : déclaration ou création via un *new*.
- implicite : expressions, paramètre par valeur (constructeur copie).

Un constructeur a la forme suivante :

```
X::X( ... )
: appels des constructeurs pour les données membres de la classe
  séparés par des ,
{
}
```

exemple :

```
TDuree::TDuree(unsigned int h=0, unsigned int m=0, unsigned int s=0)
  :heure(h), minute(m), seconde(s){}
```

Ce constructeur initialise une durée avec les arguments s'ils sont présents, ou à 0 s'ils sont absents[1].

```
TDuree d1, d2(1), d3(1, 2, 3);
```

---

[1]
```
TDuree::TDuree(unsigned int h=0, unsigned int m=0, unsigned int s=0)
  {heure=h, minute=m, seconde=s;}
```
est beaucoup plus lent : en effet les données membres sont d'abord initialisées avec leur constructeur par défaut, puis affectée. Alors qu'avec la version précédente, il sont seulement initialisés.

initialise d1 à (0, 0, 0), d2 à (1, 0, 0) et d3 à (1, 2, 3).

Il existe deux constructeurs spéciaux, qui sont fournis par le système si le programmeur ne munit pas sa classe avec des constructeurs spécifiques :

• Le **constructeur par défaut {**EX "constructeur:défaut"**}**(ou constructeur implicite)***X(){}*** est un constructeur sans argument. Ce constructeur est fourni par le système si aucun autre constructeur n'est défini par le programmeur. Le constructeur par défaut fourni par le système appelle les constructeurs par défaut de chacune des données membres (non *static*) de la classe : l'absence de définition par le programmeur du constructeur par défaut est généralement source d'erreurs de compilation :

1. Si le programmeur définit un constructeur, sans définir le constructeur par défaut, ce dernier n'existe pas :

```
class X
{ public : X(int a){}  // la classe X n'a pas de
                       // constructeur par défaut
};
class Y
{ private : X a;};     //le constructeur par défaut de X
                       // est appelé pour initialiser a

void main()
{ Y y;}
```

Le compilateur ne sait pas générer de constructeur par défaut pour la classe X. Le message est : *compiler could not generate default constructor for class 'Y'.*

2. c'est ce constructeur qui est appelé lors de l'appel du constructeur d'une classe dérivée

```
class X { public : X(int a);};
class Y : public X {};
Y y;
```

Le compilateur génère alors un message d'erreur : *cannot find default constructor to initialize base class 'X'.*

3. Il peut y avoir des problèmes d'ambiguïté, avec des constructeur ayant des paramètres par défaut :

```
class X { public : X(){} X(int i=0){} };
X x1(1), x2;
```

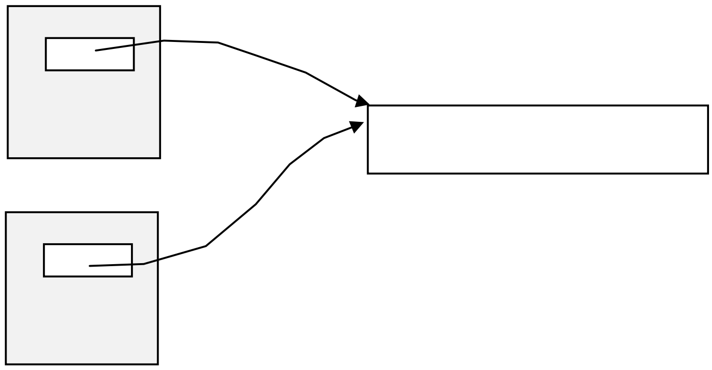
La déclaration de *x1* est OK, en revanche pour *x2* il y a ambiguïté entre le constructeur par défaut et le deuxième constructeur.

• Le **constructeur copie {EX "constructeur:copie"}*X(const X&){}*** qui est un constructeur avec un argument de type X spécifié par référence. Si le programmeur n'écrit pas de constructeur copie, le système en fournit un qui initialise la nouvelle instance en appelant les constructeurs copies pour chacune des données membres de l'instance. Ce constructeur est appelé implicitement lors du passage de paramètre par valeur, et lors du retour d'une valeur pour une fonction :

```
class X { .... };
Y f (X x)
{ X x' = x;    // création implicite d'une variable locale initialisée à x
...
```

```
}        // destruction de la variable locale
```

Un problème peut apparaître si la classe X contient des pointeurs vers des données qui lui sont propres, et qu'elle libère dans son destructeur : si le programmeur ne fournit pas un constructeur copie qui réalise la copie "en profondeur", ces zones seront libérées deux fois :



Le problème est de même nature pour le retour d'une valeur par une fonction :

```
X f( ...)
{ X x;
  ...
  return x;    // provoque une copie de x
}              // puis x est détruit en sortie de fonction
```

En règle générale, dès qu'une classe contient des *"structures profondes",* il est bon de la munir d'un constructeur copie, et dès qu'elle est susceptible d'être dérivée de la munir d'un constructeur par défaut. Il est bon aussi de penser à ne pas utiliser des paramètres par valeur si ce n'est pas nécessaire, on évitera ainsi une recopie qui peut être volumineuse en espace et en temps. D'autre part on ne peut remplacer systématiquement le type *X* de la fonction par un type *X&*, en effet dans le cas précédent, on aurait une fonction qui ferait référence à une variable locale à la fonction, qui a disparu une fois l'exécution de la fonction terminée.

Exemple d'utilisation des constructeurs :

```
class X
{ int i;
  public :   X();
             X(const X&a);
             X(int a);

};
```

```
X a;                // appel du constructeur implicite
X b(a);             // appel du constructeur copie
X d(1);             // appel du troisième constructeur
```

destructeur

Un **destructeur {**EX "destructeur" \b**}**est une fonction membre appelée chaque fois que l'instance disparaît. une instance disparaît lorsqu'elle devient hors de portée, ou lorsqu'on la supprime explicitement (*delete*). Un destructeur porte le même nom que la classe précédé d'un tilde, et n'a pas de paramètres ni de valeur retournée.

- Si le programmeur ne définit pas de destructeur, le système en fournit un qui appelle le destructeur de chacune des données membres.

- Si le programmeur définit un destructeur, le corps du destructeur est effectué, puis le système génère un appel du destructeur de toutes les données membres.

```
{ TDuree d1;        // appel du constructeur de TDuree pour d1

      { TDuree d2; // appel du constructeur de TDuree pour d2

      }                  // appel du destructeur de TDuree pour d2

}                        // appel du destructeur de TDuree pour d1
```

Dans le cas de *TDuree* le destructeur n'a rien à faire. en revanche pour une instance qui aurait acquis de la mémoire, il faudrait libérer cette mémoire.

Si le destructeur n'est pas défini par le programmeur, le compilateur en génère un qui ne fait rien.

Lorsqu'on sort d'un programme par *EXIT*, les objets de portée locale ne sont pas détruits, en revanche les objets de portée globale le sont. Lorsqu'on sort d'un programme par *ABORT*, il n'y a aucun appel de destructeur.

initialisation des données membres

{EX "donnée membre:initialisation"**}**

Lors de la création d'une instance les données membres sont initialisées dans l'ordre où elles apparaissent dans la déclaration de la classe. Ceci se fait en début d'appel du constructeur. Cette initialisation se fait explicitement ou implicitement :

```
class X
{  private :
     int t[1000];
   public :
     X();
};

X::X(){int i; for(i=0;i<1000;++i){t[i]=0;}}

class Y
{
   private :
     X x;
   public :
     Y( X & p)
        ❶: x(p){} // initialisation explicite utilisation
                  // du constructeur copie

        ❷{ x=p; } // initialisation implicite utilisation du
                  // constructeur par défaut puis
                  // de l'affectation
};
```

La version **1** prend deux fois moins de temps que la version **2**.

```
class C
{ public :
    int a; int b;
    C()
      ❶: a(1), b(a){}
      ❷: b(1), a(b){}
};
```

En **1** l'initialisation correspond aux intentions du programmeur. En revanche en **2** c'est l'appel *a(b)* qui de toute façon est effectué en premier et donc *a* est initialisé à une valeur indéterminée.

FONCTIONS MEMBRES ET AMIES.

fonctions amies.

{EX "fonction:amie"**}**

Les fonctions amies ne sont pas des fonctions membres, mais peuvent accéder à l'information privée d'une classe. Dans certains cas, les règles de masquage de l'information sont trop fortes : une fonction qui n'est pas une fonction membre a besoin d'accéder aux données privées d'une instance de la classe : on parle alors de fonctions amies.

exemple :

supposons qu'on veuille surcharger les opérateurs << et >> pour pouvoir écrire des durées, nous écririons dans la déclaration de la classe *X* :

```
friend ostream & operator <<( ostream & out, TDuree & d);
friend istream & operator >>( istream & in, TDuree & d);
```

Les définitions proprement dites seront :

```
ostream & operator <<( ostream & out, TDuree & d)
{ out<<heure<<"h "<<minute<<"m "<<seconde<<"s "; return out;}
istream & operator >>(istream & in, TDuree & d)
{ in>>heure>>minute>>seconde; return in;}
```

fonction ou donnée membre static

{EX "static:donnée membre"**}**

Les données membres ou les fonctions membres d'une classe peuvent être spécifiées en ***static*** :

Une donnée membre spécifiée en *static* est une donnée commune à toutes les instances de la classe. Une donnée membre *static* est une donnée globale qui est initialisée à 0 au chargement du programme. Elle doit être déclarée en dehors de la classe. Dans l'exemple suivant, la donnée membre *nb* sert à compter le nombre d'instances de la classe *X*.

```
class X
{ public :
    X(){++nb;}
    ~X(){--nb;}
    static int nb;
};
int X::nb;
```

Une donnée membre ***static*** existe donc en dehors de toute instance de la classe : on peut y accéder soit via une instance de la classe, soit à l'aide de l'oérateur **::** ayant le nom de la classe en premier opérande.

```
void main()
{
   X x, y, z;
   cout<<X::nb;
}
```

Les fonctions membres **static** étant indépendantes de toute instance, ne peuvent faire référence au pointeur **this**, et donc ne peuvent accéder qu'à des membres statiques de la classe.

fonction membre const

{EX "const:fonction membre"**}**

Une fonction membre peut être spécifiée *const* pour indiquer au compilateur qu'elle ne modifie pas l'instance de la classe pour laquelle elle est appelée.

exemple :

```
class X
{ public :   X():val(0){}
             a ()const { cout<<val<<"\n";}    // pas de modification
                                              // possible de val
             b (){ val = 3;}
  public :   int val;
};

void f( X & x1, const X & x2)
{ x1.a();
  x1.b();
  x1.val=0;

  x2.a();
  x2.b();   // devrait être interdit par le compilateur
  x2.val=0; // c'est interdit heureusement
}
```

Si la fonction membre est spécifiée *const* celà veut dire que le pointeur *this* est aussi *const*.

conversion de type

{EX "conversion de type"**}{**EX "type:conversion"**}{**EX "transtypage"**}{**EX "cast"**}**
Les conversions de type peuvent être explicites : **{**EX "conversion:implicite"**}**

```
float pi=3.14;
cout<<int(pi); // écrit 3
```

ou implicites :

```
void f( int a) {cout<<a;}
f(pi)
```

Il y aura conversion implicite du type *A* vers le type *B* si :

On peut définir des convertisseurs de type de deux façons différentes :

- *B* a un constructeur ayant un argument de type *A* *B::B(A)*

```
class B
{  ...
  public :
    B(A a){ ... }
}
void f( B b){ ... }
void main ()
{  A a;
    f(a);    // une instance de la classe B est crée à partir de a
             // et est transmise à la fonction f.
}
```

Il faut être prudent car les choses peuvent être beaucoup plus complexes :

```
class R
{
  private : int a; int b;
  public :
     R(int pa=0, int pb=1) : a(pa), b(pb) {}
     int operator ==(R & x) { return (a==x.a)&(b==x.b); }
```

```
};
int zero( R r){return r==0;} // 0 est converti en R avec 1 comme
                    // second argument, puis il y a comparaison avec r

void main()
{
  R r(2, 3);
  cout<<zero(r)<<" "<<zero(0); // voir commentaire précédent.
}
```

Pour éviter le surcoût dû à la conversion implicite à chaque appel de la fonction *zero*, nous écrirons :

```
int zero( R r){
   static R z(0,1);
   return r==z;}
```

- *B* a une fonction membre *A* opérateur de conversion : *operator A () {...}* (les opérateurs de conversion ne sont pas précédés d'un type lors de leur définition, le type retourné est donné par leur nom...)***{****EX "opérateur:conversion de type"****}***

```
class R
{
  private : int a; int b;
  public :
     R(int pa, int pb) : a(pa), b(pb) {}
     operator float (){return float(a)/b;}
};

void main()
{
  R r(2, 3);
  cout<<r<<"  "<<sqrt(r); // dans les deux cas r est converti en float
}
```

La multiplicité des opérateurs de conversion peut créer des problèmes :

```
class R
{
  private : int a; int b;
  public :
     R(int pa, int pb) : a(pa), b(pb) {}
     operator float (){return float(a)/b;}
     operator int(){ return int(float(a)/b) ;}
};

void main()
{
  R r(2, 3);
  cout << r; // ambiguïté : Il faut alors utiliser explicitement
                    // l'opérateur désiré float ou int
}
```

Les types prédéfinis ont leurs propres opérateurs de conversion, et les règles sont malheureusement complexes :

```
void f(long a){ cout<<"long "<<a;}
void f(int a){ cout<<"int "<<a;}

void main()
{
  f('x'); // OK converti en int
}
```

```
void f(long a){ cout<<"long "<<a;}
void f(unsigned int a){ cout<<"u int "<<a;}
```

```
void main()
{
  f('x'); // ambiguïté entre les deux fonctions.
}
```

Il vaut donc mieux dans la plupart des cas expliciter les conversions de type désirées.

surcharge des opérateurs

{EX "opérateur:surcharge"**}**

Un opérateur ❖ peut être déclaré comme fonction membre d'un classe. Dans ce cas, l'opérande de gauche est l'instance pour laquelle est appelée l'opérateur :

```
classe X
{  private : ....
    public : T operator ❖ ( ... ){ ... }
};
```

ou comme une fonction non membre :

```
classe X
{  private : ....
    public : ...
};
 T operator ❖ ( const X & x, ... ){ ... }
```

L'opérateur = est fourni automatiquement par le système (si le programmeur ne le définit pas) : l'opérateur = fourni par le système est un appel des opérateurs = pour chacune des données membres.

Tous les opérateurs peuvent être surchargés sauf ° **., .*, ::, ?:** .

Les opérateurs =, (), [], et -> ne peuvent être surchargés qu'en tant que fonctions membres d'une classe (c'est la définition du langage qui l'oblige). Pour les opérateurs unaires, on choisira généralement d'en faire des fonctions membres de la classe. en revanche pour les autres opérateurs il vaudra mieux, en général en faire des fonctions non membres.

Si l'opérateur est une fonction membre de la classe, et s'il y a des conversions implicites, la symétrie de l'opérateur peut ne pas être respectée :

```
class C
{ private : float a, b;
  public :
    C (float pa=0.0, float pb=0.0):a(pa), b(pb){}
    C operator + (const C & c);
};

C C::operator + (const C & c)
{C cl(*this);cl.a+=c.a;cl.b+=c.b;return cl;}

void main()
{
 C c(3.1,4.2);
 c=c+2.0; // légal 2.0 est converti en C(2.0, 0.0)
 c=2.0+c; // erreur de compilation illegal structure operation
}
```

Si l'opérateur n'est pas une fonction membre, il est généralement nécessaire de le déclarer en ami de la classe afin qu'il puisse accéder aux données membres privées de la classe :

```
class C
{ private : float a, b;
  public :
    C (float pa=0.0, float pb=0.0):a(pa), b(pb){}
```

```
    friend C operator + (const C & c1, const C & c2);
};

C operator + (const C & c1, const C & c2)
      {C cl(c1);cl.a+=c2.a;cl.b+=c2.b;return cl;}

void main()
{
 C c(3.1,4.2);
 c = c + 2.0; // OK 2.0 est converti en C(2.0, 0.0)
 c = 2.0 + c; // OK 2.0 est converti en C(2.0, 0.0)
 c = 2.0 + 3.0;     // dans ce cas le + est le + entre flottant.
                    // c'est le résultat 5.0 qui est converti en C
}
```

EXEMPLES.

Un exemple complet : TDuree :

```
class TDuree
{
  private:

       unsigned int heure, minute, seconde;
       unsigned long conv ( )
       {unsigned long l;l=heure*3600L+minute*60+seconde;return l; }

       TDuree  invconv( unsigned long d)
       {  heure = d/3600; d=d%3600;
          minute = d/60;  seconde = d%60; return *this;
       }

  public :

       TDuree (unsigned int h=0, unsigned int m=0, unsigned int s=0);
       ~TDuree();

       TDuree &operator =  (TDuree &d);
       TDuree operator +  (TDuree &d);
       TDuree operator -  (TDuree &d);
       TDuree operator ++ ();
       int operator <  (TDuree &d);
       int operator == (TDuree &d);
       int operator != (TDuree &d);

       friend istream& operator >>(istream& in, TDuree &d);
       friend ostream& operator <<(ostream& out, TDuree &d);
} ;
```

```
TDuree::TDuree(unsigned int h, unsigned int m, unsigned int s)
 : heure(h), minute(m), seconde(s){}
```

```
TDuree::~TDuree(){}
```

```
TDuree &TDuree::operator = ( TDuree &d)
{ heure=d.heure; minute = d.minute; seconde=d.seconde;
  return *this; }
```

```
TDuree TDuree::operator + ( TDuree &d)
{ TDuree l; return l.invconv(conv()+d.conv());}
```

```
TDuree TDuree::operator - (TDuree &d)
{ unsigned long ud1, ud2, ud;TDuree l;
  ud1=conv(); ud2=d.conv(); if(ud1<ud2) ud=0;else ud= ud1-ud2;
  return l.invconv(ud);
}
```

```
TDuree TDuree::operator ++ ()
{  TDuree l;return l.invconv(conv()+1); }
```

```
int TDuree::operator < (TDuree &d)
{ return conv()<d.conv();}
```

```
int TDuree::operator == (TDuree &d)
{ return conv()==d.conv();}
```

```
int TDuree::operator != (TDuree &d)
{ return conv()!=d.conv();}
```

```
istream& operator >>(istream& in, TDuree &d)
{ in>>d.heure>>d.minute>>d.seconde;
  d.invconv(d.conv());
  return in;
}
```

```
ostream& operator <<(ostream& out, TDuree &d)
{ out<<d.heure<<"h "<<d.minute<<"mn "<<d.seconde<<"s\n";
  return out;
}
```

```
void main ()
{TDuree d1(10), d2(2), d3(3);
     cout<<d1;
     cin>>d2>>d3;
     d1=d1-d2+d3;
     cout<<d1;
}
```

La classe TEtudiant.

Supposons qu'on veuille définir une classe TEtudiant pour représenter un étudiant. Nous voulons pouvoir faire des comparaisons et des affectations. Les comparaisons se font entre deux instances de la classe *TEtudiant* ou entre une instance de la classe *TEtudiant* et un nom d'étudiant.

```
class TEtudiant
{ private :  char * nom;
             char * prenom;
             long numero;
  public :   TEtudiant(char * n, char * p, long num);
             ~TEtudiant();
             int operator == (TEtudiant & e);
             int operator == (char * n);
             TEtudiant & operator = (TEtudiant & e);
};
```

```
TEtudiant::TEtudiant(char * n, char * p, long num)
: nom(new char [strlen(n)+1]),prenom( new char [strlen(p)+1]),
  numero ( num )
{  strcpy(nom, n);
   strcpy(prenom, p);
}
```

```
TEtudiant::~TEtudiant()
{ delete [] nom; delete [] prenom;}
```

```
int TEtudiant::operator == (TEtudiant & e)
{  return strcmp(nom, e.nom); }
```

```
int TEtudiant::operator == (char * n)
{  return strcmp(nom, n); }
```

L'opérateur d'affectation s'écrit de la façon suivante :

```
TEtudiant & TEtudiant::operator = (TEtudiant & e)
{  if(this!=&e)
   {
```

```
        delete [] nom;
        nom = new char [strlen(e.nom)+1]; strcpy(nom, e.nom);
        delete [] prenom;
        prenom = new char [strlen(e.prenom)+1]; strcpy(prenom, e.prenom);
        numero = e.numero;
    }
    return *this;
}
```

exemple : une classe chaîne

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

#if !defined _STRING
#define _STRING
class String
{
  unsigned int longueur;
  char *tete;

  public :

  String( );  String( char * s);  String( const String& s);  String( char c);
  ~String();

  friend String  operator+ (const String &op1, const String &op2s);

  String& operator=(const String &s);
  String& operator+=(const String &s);

  friend int operator <(const String &op1,const String &op2);
  friend int operator <=(const String &op1,const String &op2);
  friend int operator >(const String &op1,const String &op2);
  friend int operator >=(const String &op1,const String &op2);

  friend int operator ==(const String &op1,const String &op2);
  friend int operator !=(const String &op1,const String &op2);

  char operator []( int i);

  String substr (unsigned deb );
  String operator () ( unsigned deb );
  String substr (unsigned deb, unsigned lg);
  String operator () ( unsigned deb, unsigned lg);

  unsigned int Longueur ()const;
  friend ostream& operator<<(ostream & o, const String & s);
  friend istream& operator>>(istream & i, String & s);
} ;
#endif
```

Constructeur / destructeur

```
String::String()
:longueur(0),tete (new char [1])
{
      tete[0]=0;
} ;
```

```
String::String( char * x)
: tete ( new char [strlen(x)+1] ), longueur(strlen(x))
{
      strcpy(tete, x);
} ;
```

```
String::String(const String& s)
: longueur (s.longueur), tete (new char [s.longueur+1])
{
      strcpy(tete, s.tete);
}
```

```
String::String( char c)
longueur (1), tete (new char [2] )
{
      tete[1]=0;tete[0]=c;
}
```

```
String::~String(){
        delete  [] tete;
} ;
```

lire ecrire

```
ostream& operator<<(ostream & o, const String & s){
       o<<s.tete;
       return o;
}
```

```
istream& operator>>(istream & i, String & s){
       char * l=new char [1000];
        i>>l;
        delete [] s.tete;
        s.tete = new char [strlen(l)+1];
        strcpy(s.tete, l);
        delete [] l;
        return i;
}
```

concaténation

```
String operator + (const String& op1, const String& op2) {  //concaténation

       String l; delete [] l.tete;
       l.longueur = op1.longueur + op2.longueur;
       l.tete = new char [l.longueur+1] ;
       strcpy(l.tete, op1.tete);
       strcat(l.tete, op2.tete);
       return l;
}
```

affectation

```
String &String::operator = (const String& s)  //affectation{
       if( this!=&s){
          delete [] tete;
          longueur =  s.longueur;
          tete = new char [longueur+1] ;
          strcpy(tete, s.tete);
       }
       return *this;
}
```

```
String &String::operator += (const String& s) {
       char * l = tete;
       longueur =  longueur + s.longueur;
       tete = new char [longueur+1] ;
       strcpy(tete, l);
       if(this!=&s)   strcat(tete, s.tete);
       else           strcat(tete, l);
       delete [] l ;
       return *this;
}
```

égalités et relations

```
int operator ==(const String &op1,const String &op2){
       return !strcmp(op1.tete, op2.tete);
};
```

```
int operator !=(const String &op1,const String &op2)
```

```
{
       return !strcmp(op1.tete, op2.tete);
};
```

```
int operator <(const String &op1,const String &op2){
       return strcmp(op1.tete, op2.tete)<0;
};
```

```
int operator <=(const String &op1,const String &op2){
       return strcmp(op1.tete, op2.tete)<=0;
};
```

```
int operator >(const String &op1,const String &op2){
       return strcmp(op1.tete, op2.tete)>0;
};
```

```
int operator >=(const String &op1,const String &op2){
       return strcmp(op1.tete, op2.tete)>=0;
}
```

sous chaine

```
char String::operator [](int i){
       return tete[i];
}
```

```
String String::substr ( unsigned deb, unsigned lg){
       if (deb>strlen(tete)) return String();
       char * p=tete+deb;
       String l; delete [] l.tete;
       l.longueur = lg>strlen(p)?strlen(p):lg;
       l.tete=new char [l.longueur+1] ;
       l.tete[l.longueur]=0;
       strncpy(l.tete, p,lg);
       return l;
}
```

```
String String::operator()( unsigned deb, unsigned lg){
       if (deb>strlen(tete)) return String();
       char * p=tete+deb;
       String l; delete [] l.tete;
       l.longueur = lg>strlen(p)?strlen(p):lg;
       l.tete=new char [l.longueur+1] ;
       l.tete[l.longueur]=0;
       strncpy(l.tete, p, l.longueur);
       return l;
}
```

```
String String::substr ( unsigned deb ){
       if (deb>strlen(tete)) return String();
       char * p=tete+deb;
       String l; delete [] l.tete;
       l.longueur = strlen(p);
       l.tete=new char [l.longueur+1] ;
       l.tete[l.longueur]=0;
       strcpy(l.tete, p);
       return l;
}
```

```
String String::operator()( unsigned deb ){
       if (deb>strlen(tete)) return String();
       char * p=tete+deb;
       String l; delete [] l.tete;
       l.longueur = strlen(p);
       l.tete=new char [l.longueur+1] ;
       l.tete[l.longueur]=0;
```

```
        strcpy(l.tete, p);
        return l;
}
```

longueur

```
unsigned int String::Longueur()const
{ return longueur; }
```