Programmation Orientée Objets et C++
Première partie

Table des matières

Table des matières

Table des exemples, figures et illustrations

Bibliographie

1. C++ : Strusrutp
2. Analyse C++
3. UML
4.

Objets & classes

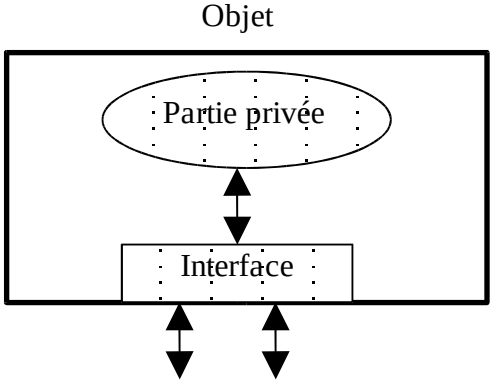
Objets

- Un objet est un morceau de données
- Contrairement aux données *passives* qui sont manipulées par des procédures, un objet peut être une donnée *active*;
- Un objet peut être un nombre, un mot, une feuille de calcul, un tableur ou l'image d'un circuit électronique;
- Si c'est un nombre, il peut savoir se doubler, calculer son opposé ou se multiplier par pi;
- On effectue ces actions en envoyant un *message* à l'objet lui disant de faire telle action. Par exemple :

dessin.afficher() on demande au dessin de s'afficher
1 + 2 on envoie le message "+" à 1 avec le paramètre 2
⇒ ***On demande à 1 de s'additionner avec 2 !***

- Les objets sont des entités privées, ils se manipulent et se modifient eux mêmes; leur interface est propre est claire;
- Ils envoient et reçoivent des messages; rien d'autre.

Encapsulation et abstraction :
- Regroupement des données et traitements dans une entité logiquement homogène
- Mise en place par les objets; elle permet de développer des applications complexes par des objets indépendants et faciles à déverminer (déboguer).



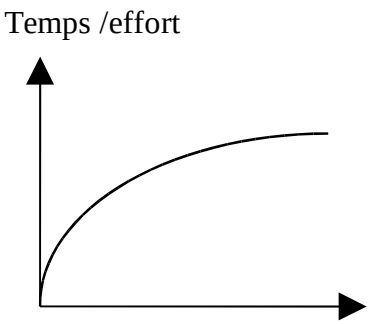
objet et interface

- Si un objet *Roue* est créé, testé et validé, ça ne sera plus nécessaire de la réinventer à chaque fois que l'on a besoin d'une roue.

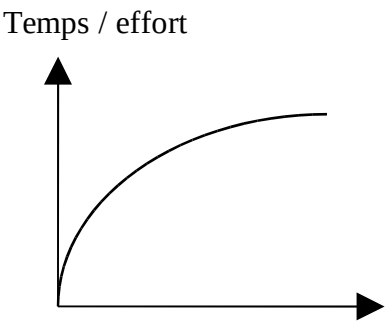
⇒ Voir par exemple les objets **string** (difficultés de char *)**, vecteur, matrice**, mais aussi **personne, étudiant**, etc. crées, validés et réutilisés.

Comparaison des coûts

Développement Objet



Développement traditionnel



Classes

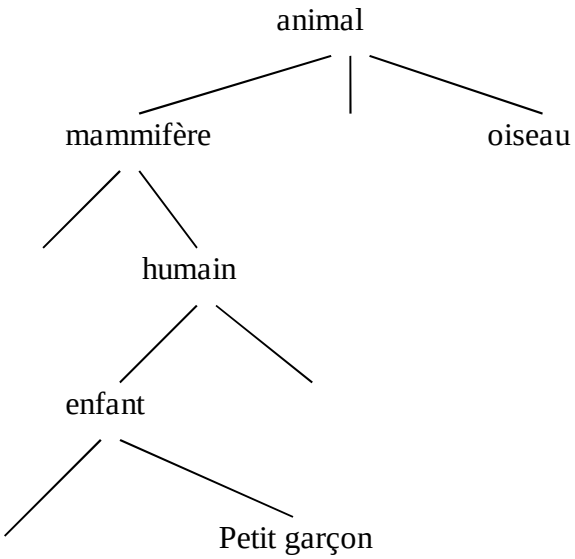
Une classe est un groupe d'objets qui partagent le même comportement et les mêmes propriétés.
Par exemple, dans la classe *animal :*

Si un animal peut respirer, manger et se reproduire, alors tous les animaux peuvent en faire autant.

On peut créer une sous classe d'objets qui possèdent des propriétés particulières. Par exemple, les *mammifères* constituent une sous classe de la classe animal avec leur propre propriétés : *les mères produisent du lait pour les petits.*

Les sous classes peuvent avoir des sous classes, par exemple, on peut passer des mammifères aux *humains*, aux *enfants* puis aux *petits garçons*, etc.

Graphe d'héritage



- Chaque sous classe *hérite* des propriétés des ses classes parents (de ses *super classes*).
- Puisque les animaux peuvent manger, les petits garçons le peuvent aussi.
- Si on ajoute une propriété à une classe, cette nouvelle capacité est héritée par toutes ses sous classes.
- Par exemple, si on dit que les animaux peuvent voler comme *super man*, alors les petits garçons héritent de ce don.

Les objets de l'univers du problème

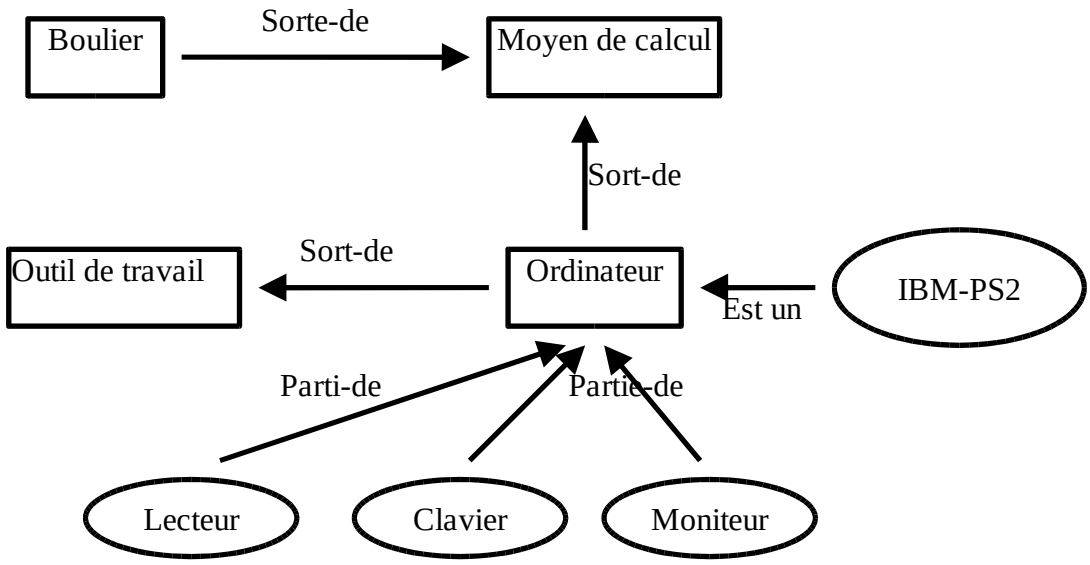
Les objets ont des propriétés que nous percevons comme des vérités premières.
Ces objets sont représentés de manière structurée faisant appel à des notions telles que :

Classification (est-un)
Rattache un objet à son type générique
Exemple : *marie est un étudiant* (l'objet marie de la classe étudiant)

Agrégation (partie-de)
Relie un objet à ses composantes
Exemple : *pied de table*

Généralisation (sorte de)
Relie les catégories d'objets à d'autres plus génériques (héritage)
Exemple : *un étudiant est une personne adulte*

Un exemple des relations intra classes: ordinateur



Si on ajoute à ce schéma : "l'ordinateur est une sorte d'objet de décoration", alors l'objet IBM-PS2 sera aussi un objet de décoration !

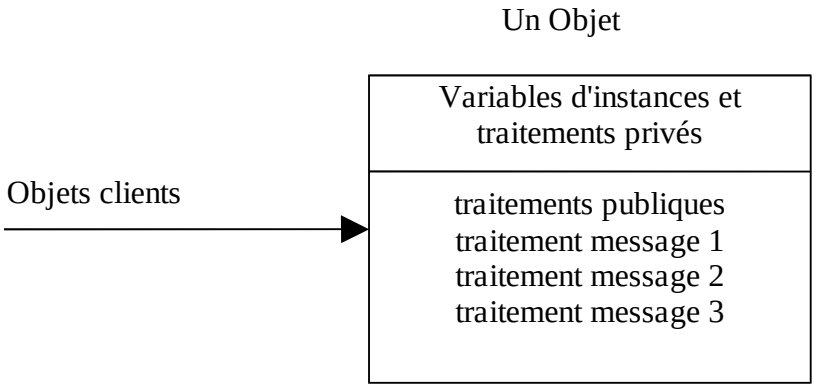
Elémenets de la programmation orientée objet (POO)

Objectifs de la POO

- Modélisation directe des objets du monde réel par des entités informatiques
- Exploitation de la redondance : dans le monde réel, on a de nombreux représentants de peu de concepts différents
- Réutilisation des composants logiciels, des composants préexistants comme en électronique, en construction automobile, etc., ...
- Réduction du couplage entre différentes parties d'une application par une interface minimal et clairement définie entre objets
- Protection et abstraction des entités par la séparation de l'interface et de l'implantation
- Réduction du coût de maintenance de logiciels

Le monde des objets : vocabulaire

Monde = collection d'objets
Objet = attributs + méthodes
Attributs = variables d'état d'objet
Méthodes = traitements / manipulation d'attributs = compétences de l'objet
Message = sélecteur + arguments
Sélecteur = indication d'une méthode de l'objet (nom de méthode)
Objet client = expéditeur du message



Principes de base de la POO

- Encapsulation

Objet = données + traitements
traitement = méthode

- Protection des données et traitements

Objet = interface + implantation

- Masquage de l'implantation

implantation fournie compilée

- Classe et instances

classe = concept = type
instance = représentant du concept = variable
factorisation des traitement dans les classes

- Communication par envoi de messages

un objet envoie un message à un autre pour lui demander d'activer une méthode

- Héritage de classes (spécialisation, généralisation)

Réutilisation et extension de classes existantes
Une sous classe d'une classe hérite des données et des traitements de la super classe.
La sous (super) classe est une spécialisation (généralisation) de la super (sous) classe.

- Polymorphisme

Manipulation de données polymorphes (multi types)
Un objet polymorphe est un objet pouvant prendre différentes formes (différents types)
Un traitement porté sur un objet polymorphe est un traitement polymorphe : le traitement peut s'appliquer invariablement sur différents types d'objets (e.g. : addition (+) des réels et des entiers)

Historique des langages objets

- Simula dans les années 60
- Smalltalk dans les années 70 (pur langage objet, définitions des 1ers interfaces graphiques)
- Langages objets natifs (purs) et langages à extension objet
  - natif : le langage ne manipule que des objets (cf. Smalltalk), Java
  - extension objet des langages existants (pratiquement tous les langages classiques) :

C	Objective C, C++
Pascal	Pascal objet
Lisp	Falvors, Xlisp, common Lisp, clos
Ada	Ada 95
Prolog	L&O, Emicat, ...

Nous allons étudier le langage C++ en suivant sa trajectoire d'évolution depuis C.

Le langage C++ : évolution de C

- Création en 1980 par B. Strustrup (Bell labs)
- Publication de la définition par exemples de C++ en 1986
- Normalisation de C++ (version 2) en 1990
- Version actuelle 3.0 (bibliothèque STL)

Le langage C++ a été crée en apportant différentes extensions au langage C.

1- Type référence à une variable

- Passage de paramètre par adresse (par référence) à une fonction :

```
void permute (int & X, int & Y)          // la présence de & spécifie un passage par adresse
{int Z = X;
 X=Y; Y=Z;
}
void main()
{int A=3, B=4;
 permute(A, B);                  // aucune différence par rapport à un passage par valeur
}
```

- Retour par référence d'une variable par une fonction

```
int & f(int & X)
{ return X; }
void main()
{int A=5;
 f(A) = 6;          // ici, la valeur de A =  6
f(A) ++ ;           // ici, A=7
}
```

La valeur retournée par la fonction **f** peut être utilisée comme un *l-value* de C (variable à gauche d'une affectation).

Remarque : dans *f*, la variable *n* ne peut pas renvoyer une variable locale car celle-ci sera détruite dès la fin de *f* et sa référence devient incohérente.

2- Paramètres par défaut dans les fonctions

Les paramètres des fonctions peuvent avoir des valeurs par défaut qui seront prises en compte si un des paramètres en question n'est pas spécifié lors d'un appel (dans l'ordre gauche droit).

```
int f(int x, int y=2, float z= 3.14) {
 ......
}
void main() {
 int t;
 t=f(5, 6, 7.5);     // 3 valeurs spécifiées, aucune valeur par défaut n'est utilisée
 t=f(7, 12);         // Z sera 3.14
 t=f(4);             // Y=2, Z=3.14
 t=f();              // ERREUR, il faut au moins un paramètre pour f
}
```

Un autre exemple de paramètres par défaut

```
enum type {noir, decafeine, expresso};
enum goût {sucre, sans_sucre, mi_sucre};

int prix_cafe(type t = noir, goût g = sucre)
{       // calcul du prix selon le type et le goût. Par défaut, un café noir et sucré
}
void main()
{int p;
 p= prix_cafe();
 p= prix_cafe(expresso);
 p= prix_cafe(expresso, sans_sucre);
p = prix_cafe(sans_sucre);          // ERREUR, il faut préciser le type d'abord
}
```

Seuls les arguments de la fin de liste des arguments peuvent avoir des valeurs par défaut.

3- Surcharge des fonctions

Fonctions de même nom mais avec des arguments différents en nombre et/ou en type.

```
void afficher(int a)       {printf("%d", a); }

void afficher(float f)     {printf("%f", f); }

void afficher(char c)      {printf("%c", c); }

void afficher(int a, int b)       {printf("%d %d", a,b); }
```

La distinction se fait selon le nombre d'arguments et leur type.

4- Opérateurs *new* et *delete*

```
int X, *Y, *Z;
Y = new int;
*Y = 45;
.....
Z = new int[5];
Z[3] = 12;
.....
delete Y;                  // libération d'un entier (en C : free(Y))
delete [] Z;                     // libération du tableau de 5 entiers occupé par Z
```

Règle d'utilisation de [] : on utilise [] dans *delete* si on s'en est servi lors de new.

5- Encapsulation

Regroupement des données et traitements en une seule entité informatique
Objet = données (attributs) + traitements (fonctions membres).

Exemple d'utilisation d'une structure

```
struct date
      {int jour, mois, an;
       void initialiser(int j, int m, int a)
       {jour = j; mois = m; an = a;}

       void lire_valeurs(int & j, int & m, int & a)
       {j=jour; m= mois; a = an;}

       void afficher()
       {printf("%d %d %d \n", jour, mois, an);}
      };

void main()
      {date hier;
       hier.initialiser(8, 2, 2000);
       hier.afficher();
      hier. Jour = 31;          // on a accès aux données encapsulées (non protégées)
      }
```

Les fonctions déclarées ici sont des **fonctions membres** et ne peuvent être appliquées qu'aux objets (variables) de type date.

Toutes les données et traitements d'une structure sont **publiques** et accessibles à tous.

6- Protection et masquage de données

Exemple de compte bancaire. Supposons qu'une banque gère ses comptes par la structure suivante :

```
struct compte1
        {int Num, Solde;
         void créer(int n, int s)  {Num=n; Solde = s;}
         int debiter(int n, int s)
         {if ((Num != n) || (s > Solde) || (s < 0)) return -1;
          solde -= n;          return Solde;
         }
         int crediter(int n, int s)
         {if ((Num != n) || (s < 0)) return -1;
          solde += n;          return Solde;}
         int bon_client()   {return (Solde > 1000);
         }
        };
void main()
        {compte1 cpt1;
         cpt1.créer(1223, 1000);
         cpt1.Solde = 2000;              // Problème d'accès non autorisées aux données
        }
```

N'importe quel client (e.g. compte1) peut modifier le solde de son compte sans passer par *débiter / créditer*.
Aussi, les clients ont accès à la fonction *bon_client* que la banque voudrait cacher (gestion interne).

Pour permettre ce genre de protections, on utilise une **classe** à la place de structure.
Remarque : un "struct" en C++ peut posséder des spécifieurs "private".

Classes : premiers exemples

L'introduction des classes permet la programmation orientée objet (POO) en C++. La suite de ce texte est consacrée à la POO avec C++.

- Une classe est un type construit analogue à une structure
- Une classe est un nouveau type défini par l'utilisateur
- Une classe permet la protection de certaines données et traitements
- Les données à protéger sont placées dans la partie (avec un spécifieur) **private** de la classe. Ces données seront uniquement accessibles par les fonctions membres de la classe.
- Le contenu d'une classe est **privé** par défaut (spécifieur "private").
- Les données et traitements de la zone **public** sont accessibles aux utilisateurs de la classe.

- Une instance d'une classe C est une variable (objet) de type C. Une instance de la classe C dispose des données et des traitements déclarés publiques dans la classe C.

- Une classe peut avoir une infinité d'instances. Les données de la classes sont dupliquées dans les instances et les fonctions sont mises en commun (via pointeurs).

Exemple de la classe compte2 (amélioration de la structure compte1)

```
class compte2 {
private :                    // section des données et traitements privés(non accessibles aux clients)
        int Num, Solde;

public:              // section des données et traitements publiques (accessibles)
        void créer(int n, int s)  {Num=n; Solde = s;}
        int debiter(int n, int s) {
          if ((Num != n) || (s > Solde) || (s < 0)) return -1;
         solde -= n;          return Solde;
        }
       int crediter(int n, int s) {
         if ((Num != n) || (s < 0)) return -1;
         solde += n;          return Solde;
        }
        int accorder_credit(int somme){
         if ((bon_client() &&(somme < 10000)) return 1;
         return 0;          // on n'accorde pas le crédit
        }

private:             // une autre section privée
        int bon_client()   {return (Solde > 1000);  }
};

void main(){
        compte2 cpt2;
        cpt2.créer(1223, 1000);
        cpt2.Solde = 2000;              // ERREUR de COMPILATION : accès interdit
}
```

Les données privées sont uniquement accessibles via les fonctions membres.
Dans une classe, on peut avoir plusieurs sections privées et publiques.
Dans la classe compte2, la fonction *bon_client* n'est accessible que par une autre fonction membre (cf. la fonction publique *accorder_credit()*).
Cette fonction est cachée par mesure de protection : la banque ne veut pas que ses critères de bon client soient découverts. De même pour les critères d'accord d'un crédit.

L'encapsulation peut également être utilisée pour cacher des données ou traitements inutiles à l'utilisateur. Ceci permet de proposer une interface propre et concis de la classe sans l'encombrer d'éléments inutiles.

Un autre exemple : classe cercle
Dans cette classe, l'utilisateur crée un cercles en fournissant le centre et le rayon mais à l'intérieur de la classe et pour des raison d'efficacité, on représente le cercle par ses deux points principaux haut_gauche (Top_left) et bas_droite (Bott_right). On prévoit donc la fonction *convertir* permettant de passer d'une représentation à l'autre.
L'utilisateur n'aura pas la possibilité d'intervenir sur ces données (même s'il les voit).

```
#include "point.hpp"          // la classe point existe déjà
class cercle
      {point Centre, int Rayon;          // private par défaut
       point Top_left, Bott_right;       // deux points utilisés dans l'affichage
       public :
       void créer(point C, int R)
       { .....
        convertir(Centre, Rayon, Top_left, Bott_right);
        .....
       }
       void dessiner()
       {// on utilise Top_left et Bott_right pour dessiner le cercle
        .....
       }
       void translater(point X)      // déplacer le centre actuel en X
       { // on recalcule les points principaux
        ...
       }

       private :
       void convertir(....) {}   // centre et rayon convertis en points cardinaux
      };
```

Ici, pour la clarté de l'interface, la procédure *convertir* (inutile à l'utilisateur) est cachée.

Un autre exemple : la classe date

```
class date {
      int jour, mois, an;
public :
       int initialiser(int j, int m, int a) {
        if (j >= 1 && j <= 31) jour = j; else return 0;
        if (m >= 1 && m <= 12) mois = m; else return 0;
        if (a > 0) an = a; else return 0;
        return 1;
       }
       void lire_valeurs(int & j, int & m, int & a) {
       j=jour; m= mois; a = an;}
       void afficher() const
       {printf("%d %d %d \n", jour, mois, an);}
};

void main() {
      date hier;
       date * ptjour = new date;
       if (hier.initialiser(8, 2, 2000) hier.afficher();
       if (ptjour -> initialiser(15, 7, 2001) ptjour -> afficher();
       delete ptjour;
}
```

Dans cet exemple, la fonction *initialiser* filtre les informations et n'accepte que des données valides.
Le mot "const" devant afficher veut dire qu'afficher ne modifiera pas les données de l'instance.

Fonctions membres inline

Une fonction *inline* est une fonction membre définie à l'intérieur d'une classe.

```
class date {
        int jour, mois, an;
public :
        int initialiser(int j, int m, int a)
        {if (j >= 1 && j <= 31) jour = j; else return 0;
         if (m >= 1 && m <= 12) mois = m; else return 0;
         if (a > 0) an = a; else return 0;
         return 1;
        }
        void lire_valeurs(int & j, int & m, int & a)
        {j=jour; m= mois; a = an;}

        void afficher() const
        {printf("%d %d %d \n", jour, mois, an);}
};
```

Les fonctions membres peuvent être définies à l'extérieur de la classe (fichier cpp).

```
class date {
        int jour, mois, an;
public :
        int initialiser(int , int , int );
        void lire_valeurs(int & , int & , int &);
        void afficher() const;
};

        int date::initialiser(int j, int m, int a)
        {if (j >= 1 && j <= 31) jour = j; else return 0;
         if (m >= 1 && m <= 12) mois = m; else return 0;
         if (a > 0) an = a; else return 0;
         return 1;
        }

        void date::lire_valeurs(int & j, int & m, int & a)
        {j=jour; m= mois; a = an;}
        void date::afficher() const
        {printf("%d %d %d \n", jour, mois, an);}
```

L'écriture ***date::*initialiser** veut dire : "fonction membre *initialiser* de la classe *date*".
L'opérateur "::" est appelé l'opérateur d'accès.
Les fonctions membres inline sont dupliquées dans les instances alors que les fonctions membres non inline sont accessibles par les instances via des pointeurs.
L'appel d'une fonction inline est plus court en temps et moins lourd que l'appel d'une autre fonction; ce qui permet une meilleure vitesse d'exécution contre une utilisation plus grande de l'espace et un degré moindre de masquage..
Par conséquent, seuls les petites fonctions doivent être inline (compromis temps/espace/masquage).

Exercices et TPs

Classe Date (les contrôles, lendemain, quantième <-> jour/mois, comparaisons,etc.). Classe Temps.

Organisation d'une application et protection des données et du code

Les langages objets permettent une protection des données qui doit être accompagnée d'un masquage de l'implantation.
On découpe en général une classe et une partie interface et une partie implantation. Seule la version binaire de la partie implantation peut être fournie; sinon, l'accès au code d'une classe permet à quiconque de modifier le code de la classe.

Exemple **:** la classe Date.
Le fichier date.hpp (ou date.h) contient la partie déclaration de la classe (**interface de la classe**).
Le fichier date.cpp (ou date.C) contient le code des fonctions membres de la classe (**implantation**).
Le fichier date.hpp est fournis aux utilisateurs pour disposer des prototypes des fonctions de la classe.
Le fichier date.cpp est compilé et on fournit à l'utilisateur date.obj (ou date.o).
Pour utiliser la classe dans un fichier "appli.cpp", on doit inclure date.hpp afin de créer des instances de la classe date et appeler les fonctions membre de la classe.
Une fois "appli.cpp" compilé, faire l'édition de liens avec date.obj (date.o) pour créer un exécutable.

Exemple date
Fichier date.hpp

```
class date{
        int jour, mois, an;
public :
        int initialiser(int , int , int );
        void lire_valeurs(int & , int & , int &);
        void afficher() const;
};
```

Fichier date.cpp

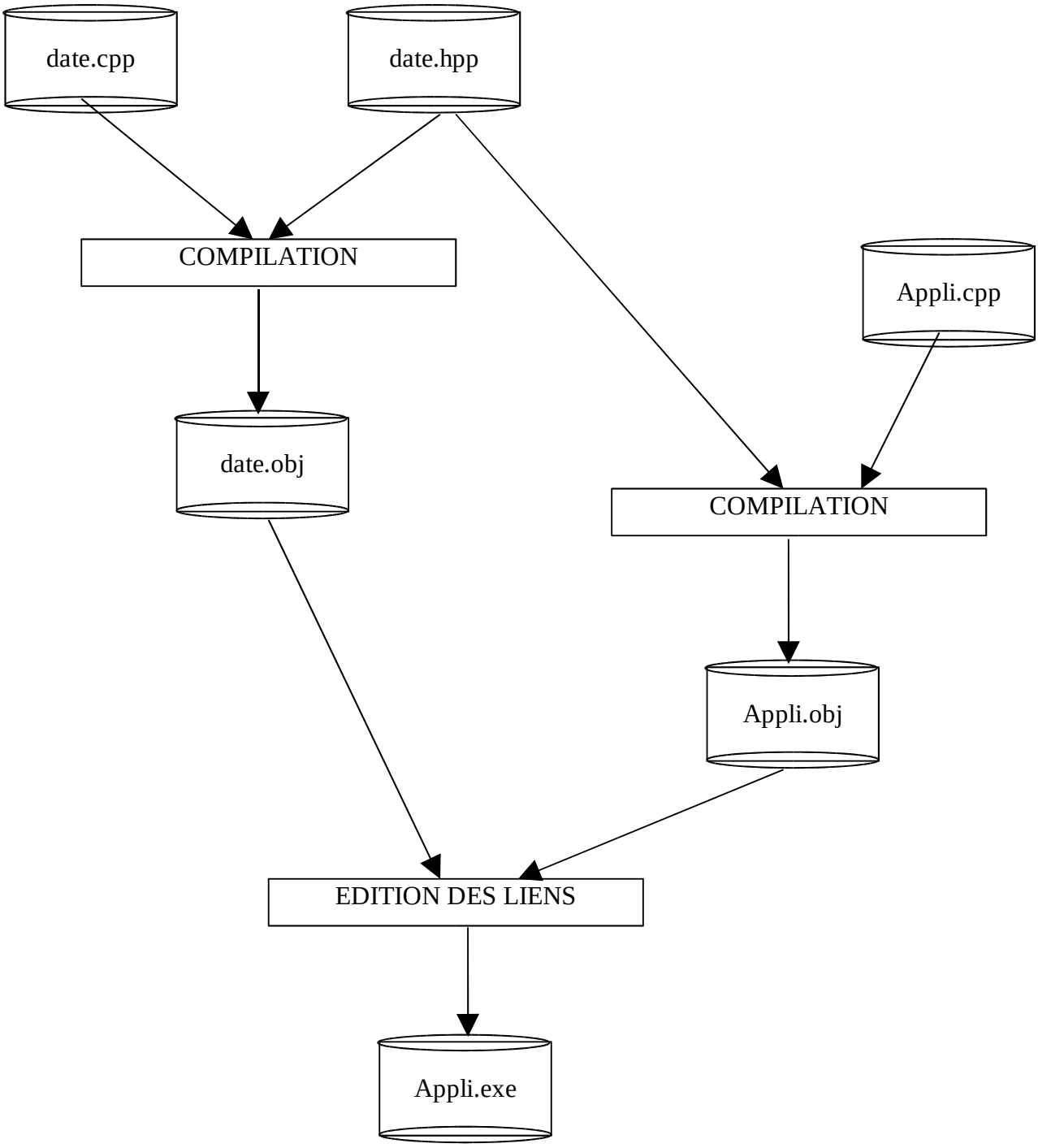
```
#include "date.hpp"
#include <stdio.h>

 int date::initialiser(int j, int m, int a)
 {if (j >= 1 && j <= 31) jour = j; else return 0;
  if (m >= 1 && m <= 12) mois = m; else return 0;
  if (a > 0) an = a; else return 0;
  return 1;
 }
 void date::lire_valeurs(int & j, int & m, int & a)
 {j=jour; m= mois; a = an;}
 void date::afficher() const
 {printf("%d %d %d \n", jour, mois, an);}
```

Fichier Appli.cpp

```
#include "date.hpp"
void main() {
 date ajourdhui;          // voir plus loin le constructeur vide
  aujourdhui.initialiser(12, 2, 2000);
  aujourdhui.afficher();
 }
```

Schéma de production de programmes

Vous pouvez réaliser ce schéma par les *projets* sous Windows ou par *Makefile* sous Linux.



Pour l'utilisation de *makefile* sous Unix, voir l'enseignant.

Exercice et TP : compléter les exemples *Date* et *Temps* (comparaisons, conversions, etc.) avec une compilation séparée.

Constructeurs et destructeurs

Constructeur

C'est une fonction spéciale (sans aucune valeur de retour) du même nom que la classe;
Elle sert à initialiser les **instances** (objets) lors de leur déclaration.
On peut avoir plusieurs constructeurs (même nom) avec divers paramètres (de 0 à n)

Fichier date.hpp

```
class date
    {int jour, mois, an;
    public :
    date(int , int , int );
     void afficher() const;
    };
```

Fichier date.cpp

```
    #include "date.hpp"
    #include <stdio.h>

    date::date(int j, int m, int a)
     {jour = j; mois = m; an = a;        }
     void date::afficher() const
     {printf("%d %d %d \n", jour, mois, an);}
```

Fichier Appli.cpp

```
    #include "date.hpp"
    void main()
    {date ajourdhui(18, 1, 2000);                    // appel automatique du constructeur
      date demain = date(12, 2, 2000);                    // idem (et non le copie constructeur)
      date hier;              // ERREUR, appel du constructeur sans argument (non défini)
    }
```

Un constructeur sans arguments (appelé constructeur vide) permet une initialisation par défaut.
On peut utiliser les paramètres par défaut dans les constructeurs.
Remarque : si on ne fournit aucun constructeur dans la classe, C++ fournit un constructeur vide (par défaut). Par contre, dès que l'on définit un constructeur, le compilateur ne fournit plus le constructeur vide.

Un exemple de constructeur sans argument pour la classe date (il ne faut pas oublier d'insérer son prototype dans date.hpp):

```
    date::date()          // par défaut on mettra la date 01/01/1901      !!
     {jour = 1; mois = 1; an = 1901;    }
```

Remarque : l'ajout de ce constructeur est en fait la surcharge des constructeurs dans la classe (voir plus loin).
Remarque : hors une initialisation immédiatement après une déclaration , on ne peut pas appeler un constructeur dans un bloc {}: **{Date d; d(12, 5, 2002);}** est une erreur mais **{Date d; d=Date(…);}** est correcte. A ne pas confondre avec les initialisations dans les sous classes.

Destructeur

Rappel : durée de vie d'une variable

```
void f()
{int X;      // X est créé à cet endroit
 date D;     // D est créé ici
 ....
}            // ici, X et D disparaissent
```

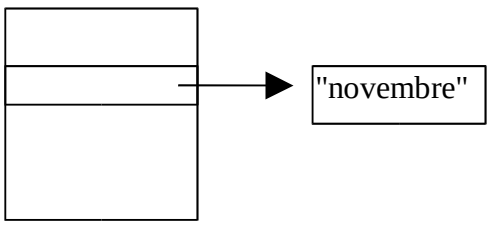
Les instance d'une classe ont la même durée de vie que les autres variables d'un type de base.
Remarquons par ailleurs que les types de base sont aussi des classes prédéfinies.

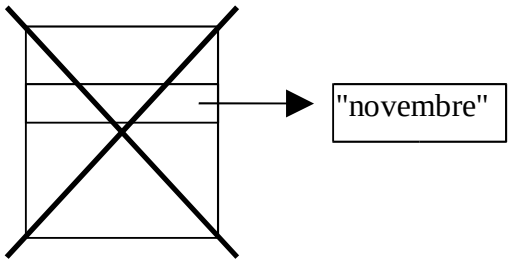
Un **destructeur** est une fonction spéciale, sans argument qui s'exécute avant la destruction d'un objet.
Le nom de destructeur est le même que celui de la classe précédé du symbole "~".
S'il est fourni, il sera appelé automatiquement lors de la suppression d'une instance (ou de l'appel de *delete* pour la destruction suite à une allocation par *new*). Ceci permet au programmeur d'intervenir pour supprimer éventuellement des zonées allouées dynamiquement, de fermer des fichiers, etc.

Exemple d'utilisation : supposons que le champ mois de la classe date est une chaîne de caractères.

```
class date
      {int jour, an;
       char * mois;
      public :
      date(int , char * , int );
      };

date::date(int j, cha * m, int a)
       {jour = j; an = a;
        mois = new char[strlen(m)+1];
        strcpy(mois, m);
       }
void main()
      {date D(1, "novembre", 2000);
       ....
      }      // l'objet D disparaît ici
```

Lors de la disparition de D, on remarque que la chaîne de caractères allouée n'est pas supprimée.





La chaîne de caractères "novembre" continue à occuper de la mémoire.**Utilisation du destructeur**

```
class date
      {int jour, an;
       char * mois;
      public :
      date(int , char * , int );
      ~date();
      };
date::date(int j, char * m, int a)
       {jour = j; an = a;
        mois = new char[strlen(m)+1];
        strcpy(mois, m);
       }
date:: ~date()
       {delete [] mois;          // les champs jour et an disparaissent automatiquement
        }
void main()
      {date D(1, "novembre", 2000);
       ....
      }      // l'objet D disparaît ici
```

Cette fois, on supprime la chaîne "novembre" avant que le reste de l'objet D disparaisse.

En règle générale, si une classe déclare des pointeurs avec de l'allocation dynamique, on doit prévoir un destructeur pour les zones allouées. Dans le cas contraire, la définition d'un destructeur est inutile (sauf si d'autres actions doivent être entreprises à la destruction d'un objet).
C++ s'occupe de supprimer les champs de type de base comme int, float, char, pointeur (mais pas l'objet pointé).

Destruction des objets pointés

Etant donné la classe date, on considère le programme d'utilisation suivant :

```
void main()
      {date D(1, "novembre", 2000);
       date * pt_date=new date(12, "juin", 1999);
      .....
      delete pt_date;
      }      // l'objet D disparaît ici
```

Lorsqu'on utilise des pointeurs sur des objets et l'on crée ces objets par *new*, il ne faut pas oublier de supprimer les objets dynamiquement créés par *delete*.
Un question peut se poser dans le cas des classes comme *date* qui contiennent un pointeur en leur sein : l'exécution de **delete pt_date** supprime-t-elle la chaîne de caractères pointée par *mois* ?
La réponse est que dans le cas des objets en C++, l'exécution de *delete* sur un objet dynamiquement créé enclenche l'appel du destructeur de l'objet en question. Donc, en conséquence de *delete pt_date*, la chaîne de caractères *mois* sera supprimée par le destructeur de l'objet pointé par *pt_date*.
Il n'est donc pas nécessaire d'essayer de supprimer la chaîne *mois* par d'autres moyens (tel qu'un appel explicite au destructeur par *date::pt_date->~date()*).

Remarque : dans le code d'une fonction, toutes les fonctions d'une classe (y compris le destructeur) peuvent être explicitement invoquées sauf les constructeurs qui sont appelés automatiquement lors de la création des objets. A ne pas confondre avec les initialisations dans l'entête d'un constructeur d'une classe composée ou d'une sous classe).

Surcharge des constructeurs

Une classe peut avoir plusieurs constructeurs. C'est un cas de figure courant.

Exemple de surcharge de constructeurs

```
class date
{int jour, an;
 char * mois;
 public :
        date(int, char *, int);          // comme 12 "janvier" 2000
        date(char *);                    // comme "12 decembre 1999"
        date(int);                       // comme 15 (du mois et année courants)
        date();                          // la date du jour de la création de l'instance
};
date::date(int j, char * m, int a)
        {// comme dans l'exemple précédent}
date::date(char * d)
        {// convertir la chaîne d en un entier (jour), une chaîne (mois) et un entier (an)}
date::date(int j)
        {// initialiser seulement le champ jour}
date::date()
        {// prendre la date du jour (date système) traité dans un exemple plus loin}

void main()
{date aujourdhui(13);
 date noel(25, "decembre", 1999);
 date fete("1 mai 1998");
 date maintenant;
}
```

Remarque sur les constructeurs :

La présence des constructeurs dans une classe revêt d'une propriété supplémentaire :pour une classe C, le constructeur avec un seul paramètre de type T introduit en fait un opérateur de conversion implicite : T -> C.

=> Préciser avec la classe string.

Conversion inverse :

Pour pouvoir convertir un objet de type C en un objet de type T, on définit dans la classe C la fonction suivante :

```
class  C
{       ….
        operator T();

};
```

=> Voir dans la partie opérateurs.

Construction par copie d'objet

Un constructeur par copie permet de copier un objet existant dans un autre. Par exemple, avec la classe *date*, on peut avoir :

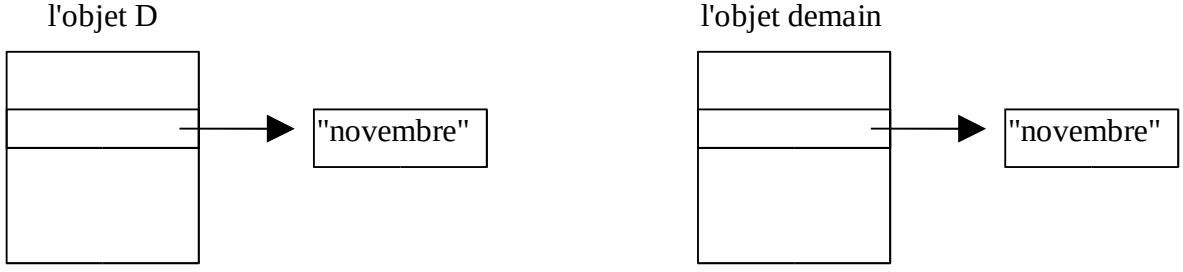
```
date D(1, "novembre", 1999);          // ici, appel de constructeur
date D1=date(2, "mars", 2000);        // idem
date demain = D;                      // ici, le copie constructeur
```

Dans l'exemple ci-dessus, le copie constructeur doit créer un nouvel objet identique à D dans l'objet *demain*.

Ecriture de copie constructeur

```
class date
      {int jour, an;
       char * mois;
      public :
      date(int , char * , int );          // le constructeur
      date(const date &);                 // le copie constructeur
      .....
      };
date::date(const date & autre_date)       // crée une copie de "autre_date"
       {jour = autre_date.jour;
        an = autre_date.an;
        mois = new char[strlen(autre_date.mois)+1];
        strcpy(mois, autre_date.mois);
       }
      .............
void main()
      {date D(1, "novembre", 2000);
      date demain = D;                    //appel du  copie constructeur
      }
```

- Le mot **const** dans le paramètre du copie constructeur veut dire : "le paramètre ne change pas".
- Le paramètre du copie constructeur doit être passé par référence.



Les cas d'utilisation d'un copie constructeur

1- création d'une copie d'un objet (comme ci-dessus)

Exemple :

```
date maintenant(12, "février", 2000);
date hier = maintenant;
```

Remarque : à ne pas confondre avec le cas suivant :

```
date hier=date(1, "janvier", 2000)
```

Ici, on utilise un constructeur de la classe date.

2- passage d'un objet en paramètre par valeur à une fonction
Exemple :

```
int f(date D)              void main()
{                  {date hier(1, "janvier", 1999);
  .....                      ....
  .....                   y = f(hier);
}                  }
```

Au moment de l'appel de la fonction *f*, l'objet *hier* est copié dans le paramètre *D*.
Dans la plupart des langages de programmation, le même mécanisme de copie est utilisé quelque soit le paramètre passé par valeur.

3- retour d'une valeur d'une fonction
Exemple : une fonction de calcul du plus grande date entre 2 dates D1 et D2. La fonction renvoie une date (la plus récente)

```
date max(const date D1, const dat D2)      void main()
{ date M;                  {date hier(1, "janvier", 2000);
  // calcul dans M de la plus grande          date demain(3, "janvier", 2000);
  // date entre deux paramètres D1 et D2      date plus_grande=max(hier, demain);
  return M;
}                  }
```

- A l'exécution de "return M", une copie de M est renvoyée dans la variable "plus_grand".
- Ce mécanisme existe aussi dans la plupart des langages de programmations.
- La présence de "const" évite toute modification des paramètres D1 et D2 dans "max".

Un exemple complet : la classe date

fichier date.hpp

```
// tableau de conversion mois (1..12) à la chaîne de caractères du mois
char * month[]={"janvier", "février", "mars", "avril", "mai", "juin", "juillet", "août",
                "septembre", "octobre", "novembre", "décembre"};
class date {
      int jour, an; char * mois;
 public :
      date(int, char *, int);          // 12 "janvier" 2000
      date(const char *);              // "12 décembre 1999"
      date(int);                       // 15 seulement le jour
      date(int, int, int);             // 15 01 2000
      date();                          // la date du jour
      ~date();                         // le destructeur
      date(const date &);              // le copie constructeur
      void afficher() const;
};
```

Le fichier date.cpp

```
#include "date.hpp"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>                              // pour le temps système

date::date(int j, char * m, int a)
      {jour = j; an = a;
       mois = new char[strlen(m)+1];     strcpy(mois, m);
      }

date::date(const char * d)                     // format : DD mois AAAA : 01 janvier 2000
      {char tampon[26], ch[5]; int i;
       jour=atoi(d);                           // s'arrête sur le 1er caractère non digit
       int l=strlen(d);                        // extraction de l'année
       for(i=0; i<4; i++) ch[i]= d[l-4+i];
       ch[4]=0; an=atoi(ch);
       strcpy(tampon, d);
       for (i=0; i<=l-3; i++)
             tampon[i]=tampon[i+3];            // décaler jours ("JJ ") en tête de tampon
       l=l-3;                                  // taille de tampon
       tampon[l-5]=0; l -= 5;                  // enlever " AAAA" de la fin (on avance '\0')
       mois=new char[l]; strcpy(mois, tampon);
      }

date::date(int j){                             // seulement le jour
      jour = j; an=0.
      mois=new char[1]; mois[0]=0;             // pour delete[] et affiche
}
```

```
date::date()                                    // prendre la date du système
        {struct tm  *area = new tm;
        time_t  t = time(NULL);
        area = localtime(&t);                   // renvoie une struct tm (voir doc)
        jour= area->tm_mday;                    // Day of month (1 - 31)
        mois=new char[strlen(month[area->tm_mon])+1];
        strcpy(mois, month[area->tm_mon]);      // Month (0 - 11)
        an=area->tm_year+1900;                  // Year (calendar year minus 1900)
        }

date::date(int j, int m, int a)         // m : 1 .. 12
        {jour=j; an=a;
        mois=new char[strlen(month[m-1])+1];
        strcpy(mois, month[m-1]);
        }
date:: ~date()      {delete [] mois; }

date::date(const date & autre_date)             // crée une copie
         {jour = autre_date.jour;
          an = autre_date.an;
          mois = new char[strlen(autre_date.mois)+1];
          strcpy(mois, autre_date.mois);
        }

void date::afficher() const
        {printf("%d %s %d \n", jour, mois, an);   }
```

Le fichier Appli.cpp (donner la trace de l'exécution)

```
void main()
{date aujourdhui(13); aujourdhui.afficher();
 date noel(25, "decembre", 1999); noel.afficher();
 date fete("01 mai 1998"); fete.afficher();
 date hier("25 fevrier 2000"); hier.afficher();
 date maintenant; maintenant.afficher();
 maintenant=fete; maintenant.afficher();                // copie constructeur
 date demain(15,2,2000); demain.afficher();
}       // ici, activation du destructeur de la classe date
```

Attribut caché this

Soit les déclarations

```
date d1; d1.afficher();
date d2; d2.afficher();
```

Comment connaître, dans une méthode telle qu'*afficher*, l'objet qui appelle la méthode ?

L'objet pour le compte duquel on appelle une méthode est représenté par **this**.
Cette attribut est un pointeur sur l'objet appelant.
On utilise **this** rarement.

Un exemple d'utilisation de l'attribut this

```
class thing
{int X, Y;
 public :
 thing() { ..... };                          // constructeur
 thing(const thing & th) { .... }            // copie constructeur
 thing myself() const;
};

thing thing::myself() const
{printf("la fonction myself renvoie l'objet qui l'appelle");
 return *this;                  // this est un pointeur, *this est l'objet pointé
}

void main()
{thing X;
 thing Y=X.myself() ;           // Y reçoit une copie de X (équivalent à thing Y = X;)
}
```

Remarquons que la fonction *myself* utilise le copie constructeur pour renvoyer un objet *thing*.
Nous verrons plus loin un cas d'utilisation de **this**.

L'opérateur d'accès ::

- Comme on l'a déjà vu, cet opérateur permet de désigner une fonction membre d'une classe sous la forme : *nom_classe**::**nom_fonction(..)*

- Il permet également de désigner les attributs des classes. On s'en sert parfois pour lever les ambiguïtés dans les noms.

Exemple :

```
int mois;                          // (1) un "mois" global
class date{
      int jour, mois, an;          // (2) un "mois" dans la classe date
 public :
      date(int j, int mois, int a) // (3) un "mois" en paramètre"
      {jour = j;
       date::mois = mois;          // le "mois" de date = le "mois" en paramètre
       ::mois = mois;              // le "mois" global = le "mois" en paramètre
      }
};
```

Les fonctions membres, externes et amies

A ce stade, il ne faut pas oublier qu'en dehors des classes et de leurs fonctions membres, on peut définir des fonctions classiques (appelées **fonctions externes**) qui peuvent coexister avec les

fonctions membres. Seulement, ces fonctions ne peuvent pas manipuler les attributs privés des classes.

Pour que des fonctions externes (à la classe) puissent avoir accès aux données privées de la classe, on peut les déclarées **fonctions amies**.
C'est à dire, à l'intérieur d'une classe, on peut déclarer amie une fonction ordinaire qui va pouvoir manipuler les attributs de la classe.

Les objectifs principaux de cette déclaration peuvent être énoncés comme suit :
1- Une fonction membre possède un argument caché **this** qui lui est transmis lors d'un appel. Pour la clarté de l'explication, on dira que ce paramètre est le premier paramètre transmis.

Par exemple, dans :

```
void date::ajouter(int J);          // ajouter J  jours à *this

Utilisation :
date D(10, "mars", 2002);
int X=12;
D.ajouter(X);          // ajouter 12 jours à D
```

Ici, la date D est transmise à la fonction *ajouter* par la référence *this*. Le profile (TDA) de la fonction est : *ajouter : Date x int -> date*
Si on a besoin de surcharger la fonction *ajouter* pour pouvoir écrire :

```
friend Void ajouter(int J, date & D);     // ajouter J  jours à D

Utilisation :
date D(10, "mars", 2002);
int X=12;
X.ajouter(D);
```

Dans ce cas, le paramètre *this* ne nous intéresse plus. On veut dans ce cas le profile :

ajouter : int x date -> date

On remarque que la déclaration de la fonction surchargée *ajouter* n'est pas précédée de *date::* mais du mot clé *friend*.

2- Si une classe préexiste (sous forme binaire) et on veut lui fournir une fonction supplémentaire, on utilisera également le mécanisme de fonction amie. C'est en particulier le cas de la classe *ostream* utilisée pour les entrée-sorties (voir plus loin).

3- Il existe aussi des raisons d'efficacité d'exécution (voir plus loin).

Pour une fonction amie, **tous** les paramètres doivent être explicités (par de paramètre implicite).
Dans une fonction amie, le premier paramètre (opérande) est **toujours** l'objet pour lequel la fonction est appelée.
Lorsque le **premier** opérande n'est pas un objet de la classe, on sera obligé d'utiliser une fonction amie.

Exemple récapitulatif utilisant différents types de fonctions (membre, inline, externe)
Dans l'exemple de date étendu suivant, on utilise les différents types de fonctions : membre, inline et non inline, externes, …

fichier appli.cpp

```
#include "date.hpp"
void main()
{       date hier(12, 2, 2000);
        date aujourdhui;
        date anniversaire(15, 3, 1999);
        if ( plus_grand(hier, anniversaire)) { … }
        anniversaire.lendemain();

        anniversaire.afficher();
}
```

fichier date.hpp

```
class date {
        int jour, mois, an;
 public :
        date()                  // fonction inline. Date par défaut = 01/01/1901
                {jour =1; mois = 1; an = 1901;}
        date(int, int, int);            // constructeur définie hors l'interface de la classe
        friend date& difference(const date, const date);        // différence entre deux dates
        void afficher();
        void lendemain();
        ....
};
```

fichier date.cpp

```
#include "date.hpp"
#include <stdio.h>

date::date(int j, int m, int a)         // constructeur défini hors la classe
        { ..... }


date& difference(const date D1, const date D2)      // fonction amie ayant accès aux données
        {//calculer une nouvelle date résultat de D1 – D2
        }

bool  plus_grand(const date D1, const date D2)      // fonction externe
        { // renvoie true si D1 est plus récente que D2.
         //Utiliser la fonction amie difference et obtenir D3 = D1 – D2
        // si D3 est positive ou nulle, alors D1 est plus grande que D2
        }
```

```
inline void date::afficher()   // fonction définie hors la classe mais rendue inline
        {// affichage d'une date
        }

void date::lendemain()
        {//modifier *this en y  ajoutant 1
        }
```

On remarque que lors de la définition de "*difference*" on n'a pas utilisé "**date::**" nécessaire pour les fonctions membres de la classe *date*.

On peut penser que les fonctions amies pourraient être de simples fonctions membres. Ce n'est pas toujours opportun comme on peut le voir dans l'exemple suivant (raison d'efficacité).

Soit une classe *vecteur* et une classe matrice.

```
class vecteur
{int V[4];
 public :
 int elem(int i)            // accès au Ième élément
        {if (i<4) return V[i]; else return -9999;}
};

class matrice
{int M[4][4];
 public :
 int elem(int i; int j)     // accès au Jème élément de la Ième ligne
        {if (i<4 && j<4) return M[i][j]; else return -9999;}
};
```

supposons que l'on veuille savoir si un vecteur correspond à une ligne d'une matrice. La fonction **compare** ci-dessous effectue cette comparaison. Noter que *compare* ne peut pas être membre des deux classes. C'est pourquoi elle est externe aux classes dans la 1ère version ci-dessous :

```
int compare(const matrice & Mat, const vecteur & Vect)
{ // ligne par ligne de la matrice Mat, il faut comparer les éléments avec ceux du vecteur Vect
  // les comparaisons (16 au plus dans cet exemple) des éléments se font par des
  // appels à Mat.elem[i][j] et Vect.elem[k]
}
```

Il est évident que cette méthode d'accès n'est pas optimale. Il faudrait pouvoir accéder aux éléments de la matrice et du vecteur d'une façon plus rapide sans devoir passer par la fonction **elem**.

On déclare pour cela la fonction **compare** comme amie des classes matrice et vecteur. Rappelons que compare ne peut pas être membre des deux classe à la fois. Elle doit donc être déclarée amie des deux classes.

```
class matrice;
class vecteur
{int V[4];
 public :
 friend int compare(const matrice &, const vecteur &);
};
```

```
class matrice
{int M[4][4];
 public :
friend int compare(const matrice &, const vecteur &);
};

int compare(const matrice & Mat, const vecteur & Vect)
{// on peut maintenant accéder directement à Vect.V[i] et à Mat.M[i][j]
}
```

Classe et fonction amie

En généralisant la notion de fonction amie, on peut déclarer une classe amie d'une autre classe. De cette façon, les fonctions membres de la classe amie ont accès à la partie privée de la classe.

```
class C1
{private :
        .....
 public :
        ...
};

class C2
{private :
        .....
 public :
        ...
        friend C1;          // la classe C1 est amie de la classe C2
};
```

On peut également déclarer une des fonctions membres d'une classe comme amie d'une autre classe. De cette manière, la fonction membre déclarée amie aura accès aux données privées de la classe.

```
class C1
{private :
        .....
 public :
        int f(int X) { ... }
};

class C2
{private :
        .....
 public :
        ...
        friend C1::f(int);          // la fonction f de la classe C1 est amie de la classe C2
};
```

Surcharge des opérateurs

La notion de surcharge (redéfinition) des opérateurs existe déjà dans les langages de programmation.

```
int a, b = 4, c = 2;
double ad, bd = 3.4, cd = 2.5;
a = b + c;              // addition des entiers
ad = bd + cd ;          // addition des réels
```

L'opérateur '+' est utilisé pour les entiers et pour les réels. On dit que '+' est **surchargé**.

On peut surcharger les opérateurs en C++. Pour cela, on surcharge un opérateur portant déjà sur des types de bases pour qu'il s'applique aux nouvelles classes que l'on crée.

Exemple : si on définit la classe des nombres complexes, on préférerait pouvoir utiliser les opérateurs '+', '-' , '=' et autres sur les instances de cette classe (au lieu d'utiliser les fonctions telles que add, sub, affect, ...).

Par exemple, on préfère pouvoir écrire :

```
complexe a, b(3.4, 5), c(2.3, 8.2);
a = b + c;                      // au lieu de affect(a, add(b, c))
```

Pour cela, il faut pouvoir surcharger les opérateurs '=' et '+' dans la classe complexe.

Les règles de surcharge des opérateurs

- On ne peut surcharger que les opérateurs existants dans le langage. On ne peut pas en inventer d'autres. On ne peut donc pas surcharger par exemple '@' qui n'est pas un opérateur.

- L'opérateur surchargé conserve sa règle de priorité, d'arité (nombre d'opérandes) et d'associativité. Par exemple, si '+' est surchargé, il reste binaire (A+B), moins prioritaire que '*' (A+B*C), de priorité égale à celle de '-' et associative à gauche (A+B+C=(A+B)+C).
Remarquons qu'il existe en C/C++ des opérateurs unaires et binaires.

- On ne peut surcharger les opérateurs qu'à l'intérieur d'une classe. Au moins un des opérandes doit être instance de la classe.

Mise en œuvre de la surcharge des opérateurs

Pour par exemple surcharger '+', on déclare une fonction de la forme *operator+(...)*
La fonction qui résulte de la surcharge d'un opérateur peut être membre ou amie de la classe.

Remarque : dans la cas des fonctions (opérateurs) membres, il ne faut pas oublier que l'objet pour lequel on appelle la fonction est transmis par **this**. Ceci est vrai pour toute fonction.

Exemple :
dans la classe date, la fonction *afficher* avait été définie sous la forme *void afficher();*
Or, afficher est un-aire (on affiche le contenu d'un objet date). Mais on a bien observé que l'objet en question n'était pas passé en paramètre d'*afficher().*

Par exemple, dans :

```
date D(1, 2, 1999);
D.afficher();
```

l'objet D est transmis par *this à l'intérieur d'*afficher()*.

Il en résulte qu'en C++, l'objet pour lequel on appelle une fonction membre est présent (pointé par this) et il n'est donc pas nécessaire (ni correct) de le passer en paramètre.

Opérateur binaire avec une fonction membre

Exemple de surcharge l'opérateur '+'

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);                   // constructeur
        complexe operator+(const complexe &);       // surcharge de l'opérateur +
        complexe operator+(const float );           // ajoute d'un réel à la partie réelle

        ...
};

complexe complexe::operator+(const complexe & C)
{complexe R = *this;
 R.reel += C.reel; R.imaginaire += C.imaginaire;
 return R;
}

complexe complexe::operator+(const float F)
{complexe R(F, 0);
 return *this + R;        // on réutilise l'opérateur '+' précédent
}

void main()
{       complexe X(3.2, 7.8), Y(2.5, 7), Z;
        Z = X + Y;
        Z =Z + 1.5;
}
```

- On remarque encore une fois que pour cet opérateur binaire, on précise seulement un seul paramètre. Dans *Z=X+Y*, l'opérande gauche dans l'addition est X représenté par **this**.

- La présence de const devant le paramètre de l'opérateur signale que ce paramètre ne sera pas modifié. La présence de & est ici une simple mesure d'économie dans le passage des paramètres. On évite ainsi de copier le paramètre lors de l'appel (on passe seulement une référence sur celui-ci).

- L'opérateur '+' est surchargé pour permettre l'addition d'un réel à un complexe.

Définition de l'opérateur '+' avec une fonction amie

Dans l'exemple précédent, nous avons surcharge l'opérateur '+' pour pouvoir ajouter un réel à un *complexe*.
Rappelons également l'une des raisons de définition des fonctions amies : surcharger les fonctions (ici des opérateurs) avec des profiles différents.

Si pour la classe *complexe*, on veut les profiles suivants pour '+' :
+ : complexe x complexe -> complexe
+ : complexe x réel -> complexe
+ : réel x complexe -> complexe

Les deux premiers ont été défini dans l'exemple précédent (où on a utilisé l'argument *this*). Pour le troisième, nous devons utiliser une fonction amie :

Exemple : fonction '+' amie

```
class complexe {
        double reel, imaginaire;
 public :
        complexe(double, double);                   // constructeur
        complexe operator+(const complexe &);       // surcharge de l'opérateur +
        complexe operator+(const float );           // ajoute d'un réel à la partie réelle

        friend complexe operator+(const float, const complexe &);
                ...
};

complexe complexe::operator+(const complexe & C)
{       // voir l'exemple précédent
}

complexe complexe::operator+(const float F)
{       // voir l'exemple précédent
}

complexe operator+(const float F, const complexe & C ) // pas de complexe::
{return C+F;}           // on inverse les paramètres et on réutilise 'complexe + réel'

void main() {
        complexe X(3.2, 7.8), Y(2.5, 7), Z;
        Z = X + Y;          // complexe + complexe
        Z = Z + 1.5;        // complexe + réel
        Z = 1.7 + Y ;       // réel + complexe
}
```

Opérateur un-aire avec une fonction membre

Avec règles de définition de fonctions , un opérateur un-aire réalisé par une fonction membre n'aura aucun paramètre explicite. Son seul paramètre représenté par *this* est l'objet qui envoie le message.

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);    // constructeur
        complexe operator++();       // surcharge de l'opérateur ++ (préfixé)
        ...
};
complexe complexe::operator++()
{reel ++; imaginaire++;
 return *this;
}
void main()
{       complexe X(3.2, 7.8);
        ++X;
}
```

Remarques et explications :
- L'opérateur ++ (pré incrémentation) est un opérateur unaire.
- On remarque l'absence de paramètre dans la définition de cet opérateur. En fait, lorsqu'on écrira :

```
        complexe X(3.2, 7.8);
        ++X;
```

L'objet X est lui même transmis (via this) à la fonction d'incrémentation (comme dans la fonction afficher() de la classe Date). A l'intérieur de l'opérateur, ce sont donc les champs *reel* et *imaginaire* de X qui seront incrémentés. L'opérande de '++' est X lui même.
- L'opérateur ++ renvoie un nombre complexe. C'est tout à fait naturel de renvoyer l'objet incrémenté. Pour pouvoir écrire **Y=X++;** cet objet (X dans l'exemple ci-dessus) est présenté par *this, d'où "return *this"

Opérateur un-aire avec une fonction amie

La seule différence est que dans le cas de toute fonction amie, tous les paramètres, en particulier celui représenté par *this* doivent figurer en argument de la fonction.
L'exemple suivant démontre ceci (pas de raison particulière à cette définition dans une classe).

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);
        friend complexe operator++(complexe &);      // surcharge de l'opérateur ++ (préfixé)
        ...
};
complexe operator++( complexe & C)                   // on remarque l'absence de complexe::
{C.reel ++; C.imaginaire++;
 return C;
}
```

Dans un appel comme *++X*, l'opérande X est transmis dans le paramètre C à la fonction amie d'incrémentation .

- Rappel : dans une fonction amie, le premier paramètre (opérande) est **toujours** l'objet pour lequel la fonction est appelée.
- On peut s'interroger sur l'intérêt des opérateurs amis. Il est vrai que l'on préfère les fonctions (opérateurs) membres aux amies. Ceci est vrai en particulier lorsque le premier opérande est un objet de la classe. Mais on verra plus loin que dans certains cas, on sera obligé d'utiliser des fonctions amies notamment dans le cas de surcharge des opérateurs des classes existantes.
En particulier, lorsque le premier opérande n'est pas un objet de la classe, on sera obligé d'utiliser une fonction amie. Il y a également d'autres cas d'utilisation des fonctions amies (voir plus haut dans les nombres complexes).

Remarque à propos de l'opérateur ++

Nous savons qu'en C/C++, l'opérateur '++' peut être utilisé pour pré ou post incrémenter une valeur. Par exemple, dans :

```
T[i++] = 12;
T[++i] = 12;
```

les résultats ne seront pas les mêmes. Dans le premier cas, l'indice i est post incrémenté (incrémenté après l'utilisation de sa valeur) alors que dans le deuxième cas, l'indice i est pré incrémenté (on l'incrémente puis on l'utilise).
Il faut donc pouvoir distinguer entre ces deux cas lorsqu'on veut surcharger l'opérateur '++' (et '—').
Le problème est résolu en C++ de la façon suivante :
- Pour surcharger la pré incrémentation, on procède comme dans le cas des nombres complexes ci-dessus.
- Pour surcharger la post incrémentation, on doit se servir d'un paramètre arbitraire supplémentaire.

Exemple :

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);    // constructeur
        complexe operator++(int);    // surcharge de l'opérateur ++ (postfixé)
        ...
};
complexe complexe::operator++(int x)
{complexe C=*this;     // sauvegarder *this car c'est une post incrémentation
 reel ++; imaginaire++;
 return C;
}
void main()
{       complexe X(3.2, 7.8);
        X++;
}
```

Remarques :
- Etant donné la signification de X++ (utiliser puis incrémenter), on se voit obligé d'utiliser la variable temporaire C dans la définition de l'opérateur.
- La présence du paramètre (non utilisé) x signale au compilateur C++ qu'il s'agit de la version postfixée (post incrémentation) de l'opérateur '++'.

Définition de l'opérateur '+='

L'opérateur '+=' pour la classe complexe.
Lors de la définition d'un opérateur, on doit bien connaître le comportement et les caractéristiques de celui-ci. Par exemple, nous savons que dans X += Y, la signification est X = X+Y, et donc la valeur de X se modifie (on y ajoute la valeur de Y)mais pas celle de Y; que '+=' est binaire, etc. (voir ci-dessous).

Exemple de l'opérateur '+='

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);                       // constructeur
        complexe operator+=(const complexe &);          // surcharge de l'opérateur +=
        ...
};

complexe complexe::operator+=(const complexe & C)
{reel += C.reel;                // la partie reel de this
 imaginaire += C.imaginaire;
 return *this;
}

void main()
{       complexe X(3.2, 7.8), Y(2.5, 7), Z;
        X += Y;
}
```

Esquisse d'une méthode de définition d'opérateurs

Lors de la définition d'un opérateur, il faudra considérer les aspects suivants :
1- arité : combien d'opérandes (0, 1, 2)
2- associativité : gauche, droite
3- ce qui se modifie (*this / autres opérandes modifiable ?)

Par exemple,
- dans la définition de l'opérateur '++', on modifie et renvoie *this alors que dans '+', les opérandes ne changent pas.
- dans '=', l'opérande *this (lvalue) sera modifié mais pas l'opérande droite (le deuxième).
- dans '+=', * this sera modifié même si on réutilise '+' (pas de modification des opérandes) et '='.

4- L'opérateur peut (doit) être **ami** ? En général, il doit l'être si son première paramètre n'est pas un objet de la classe courante.

=> Voir l'exemple de '+=' ci-dessus et '=' (plus loin) et étudier ces points.

Utilité des opérateurs (fonctions) amis

Rappels :

- Dans le cas d'un opérateur un-aire, on peut utiliser indifféremment une fonction membre ou amie.
- Dans le cas d'un opérateur binaire, on ne peut utiliser un fonction membre que si le premier opérande est du type de la classe.
- **Pour la surcharge d'un opérateur binaire avec un premier opérande de type quelconque et le deuxième opérande du type de la classe, on doit utiliser une fonction amie de la classe.**

- Exemple : si l'on veut pouvoir écrire :

```
complexe X(2.3, 6.9), Y;
double D;
Y = D + X;
```

Dans ce cas, le premier opérande n'est pas du type complexe. On utilise donc une fonction amie.

```
class complexe
{       double reel, imaginaire;
  public :
        friend complexe operator+(const double , const complexe &);
        ...
};
complexe operator+(const double D, const complexe & C)
{complexe R = C;
 R.reel += D;                   // seule la partie reel est modifiée
 return R;
}
void main()
{       complexe X(3.2, 7.8), Y; double D=3.7;
        Y = D + X;                      // double x complexe => complexe
}
```

Remarque : si l'on veut écrire

```
complexe X(2.3, 6.9), Y;
double D;
Y = X + D;
```

C'est à dire, le profil de '+' : *complexe x double => complexe*
On pourra définir une fonction membre dont le seul paramètre sera un double.

Supposons que l'opérateur déclaré ci-dessus existe. On va s'en servir ci-dessous :

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);               // constructeur
        complexe operator+(double);             // complexe + double => complexe
        ...
};

complexe complexe::operator+(double D)
{return D + *this;              // appel de l'opérateur '+' ci-dessus
}
```

```
void main()
{       complexe X(3.2, 7.8), Y;
        double D;
        Y = X + D;
}
```

L'opérateur =

Etude le l'opérateur :

- Dans *A = B*, l'opérateur '=' doit libérer les zones de données éventuelles associées à A avant de lui affecter B. Donc, dans '=', on effectue, si nécessaire, une destruction de l'opérande gauche.
- Comme en C/C++, il faut pouvoir écrire *A=B=C (A=(B=C));* d'où la valeur de retour.

Surcharge de l'opérateur '='

```
class complexe
{       double reel, imaginaire;
  public :
        complexe(double, double);                       // constructeur
        complexe& operator=(const complexe &);
        ...
};

complexe & complexe::operator=(const complexe & C)    // rien à détruire avant '='
{if (this != &C)          // cas de A=A !! On ne fait rien
        {reel = C.reel;
         imaginaire = C.imaginaire;
        }
 return *this;
}
void main()
{       complexe X(3.2, 7.8), Y(2.1, 5), Z;
        Z = X = Y;
}
```

Remarque : dans *complexe X(1.2, 6.8);*
complexe Y=X;
On n'utilise pas '=' mais le copie constructeur.

Exercice : modifier la classe date pour réaliser les opérateurs suivants :
+ : date x date -> date
+ : date x int -> date
+ : int x date -> date

Remarque sur les opérateurs et la conversion de type

Dans la section traitant des constructeurs, nous avons vu que dans une classe C, un constructeur acceptant un paramètre de type T jouait le rôle d'un convertisseur du type T vers le type C.

Dans une classe C, pour pouvoir convertir un objet de type C en un objet de type T, on définit la fonction **operator T();**

Exemple de conversion de type

```
class X {   …
            public :
            X(int);             // constructeur : conversion int -> X
            void f(const X);
            operator int();     // convertisseur X -> int
        };

class Y {   …
            public :
            Y(X);                       // constructeur : conversion X -> Y
            operator X();               // convertisseur Y -> X
        };

void main()
{X a(12);               // OK,
  a.f(15);              // OK. Conversion de int vers X puis appel de f
  Y c = X(14);          // OK. Mais
  Y b = 10 ;            // Erreur : Y(X(10)) n'est pas essayé car nécessité deux
                        // conversions 10 -> X -> Y : int -> Y     non accepté
  X d = c;              // c de type Y converti en X
}
```

De même :

```
void f(X c)
{int i= int(c );        // ok, 'c' convertie de X en int grâce au convertisseur operator int();
  i = int(c );          // idem
  i = c;                // idem. Dans les 3 cas, conversions faites.
  if ( c) { …}          // c considéré comme un entier (ici bool).
}
```

Un exemple complet : la classe vecteur d'entiers

Classe vecteur

```
#include <stdio.h>
#define max_elements 20                    // au plus 20 éléments

class vecteur {
        int nb_elements;                   // nombre d'éléments dans vecteur
         int * t;
 public:
         vecteur() ;
         vecteur(const vecteur&);          // copie constructeur
         vecteur& operator=(const vecteur&);
         bool operator==(const vecteur&);
         bool operator!=(const vecteur&);
         vecteur& operator+=(int);
         vecteur operator+(int);
         const int recherche(int) const;   // recherche du ième élément
         vecteur operator+(const vecteur); // Fusion non triée
         ~vecteur() ;
         void afficher();
         void lire();
};
```

```
#include "vecteur.hpp"
vecteur::vecteur()
        {nb_elements=0;
         t=new int[max_elements];
        }

vecteur::vecteur(const vecteur& V)         // copie constructeur
        {nb_elements=V.nb_elements;
         t=new int[max_elements];
         for (int i=0; i<nb_elements; i++) t[i]=V.t[i];
        }

vecteur& vecteur::operator=(const vecteur& V)
        {if (this == &V) return *this;
         this->~vecteur();                 // détruire avant '='. Ici inutile sauf si agrandir
         nb_elements=V.nb_elements;        // Utile si on agrandit le vecteur
         t=new int[max_elements];
         for (int i=0; i<nb_elements; i++) t[i]=V.t[i];
         return *this;
        }
bool vecteur::operator==(const vecteur& V)
 {if (nb_elements != V.nb_elements) return 0;
  for (int i=0;  i<nb_elements; i++)
        if (t[i] != V.t[i]) return 0;
  return 1;
 }
bool vecteur::operator!=(const vecteur& V)
        {return !(*this == V);}
```

```
vecteur& vecteur::operator+=(int X)
 {if (nb_elements < max_elements) t[nb_elements++]=X;
  else perror("la table est pleine\n");
  return *this;
 }

vecteur vecteur::operator+(int X)
 {vecteur V = *this;   V += X;   return V;}

const int vecteur::recherche(int i) const              // recherche du ième élément
 {if ((i<0) ||(i>=nb_elements))
       {perror("mauvais indice"); return -9999; }
  return t[i];
 }

vecteur vecteur::operator+(const vecteur & V)          // FUSION non triée
  {vecteur V1=*this;
   for (int i=0; i<V.nb_elements; i++) V1 += V.t[i];       // ou V[i] si '[]' défini
   return V1;
   }

vecteur::~vecteur()     {delete [] t; nb_ele=0; }

void vecteur::afficher()
 {printf("<%d>",nb_elements);
  for (int i=0; i<nb_elements; i++) printf("%d, ",t[i]);
  putchar('\n');
 }

void vecteur::lire()
 {printf("donner %d entiers :",max_elements);
  for (int i=0; i<max_elements; i++) scanf("%d", &t[i]);
 }
```

```
void main()
{vecteur V1;                                  // constructeur vide
 V1 += 1;  V1 += 3; V1 += 2;                  // opérateur '+='
 vecteur V2=V1;                               // copie constructeur
 vecteur V3; V3=V1;                           // opérateur '='
 V1= V1 + 10;                                 // opérateur '+' et '='
 vecteur V4 = V1 + 2 + 15 ;                   // opérateur '+' et copie constructeur
 if (V1 == V4) printf("%s", "égalité de V1 et V4 \n");
 printf("le 3eme élément de V1 est = %d \n",V1.recherche(3));
 printf("%s", "la fusion de V1 et V3 donne :"); (V1+V3).afficher();
 vecteur V5; V5.lire(); V5.afficher();
}
```

Opérateur []

Dans le cas de certaines classes comme la classe *vecteur*, il est pratique de définir l'opérateur '[]' permettant d'accéder à un élément du vecteur par un indice ou par une valeur (accès associatif). Ce qui permet une écriture plus simple.

Par exemple, on pourrait écrire :

```
vecteur V; int X;
....
X=V[5];              // le 5ème élément de V (au lieu de "recherche")
X=V.eleme(5);        // idem
X=V.recherche(5);    // idem
X=V.tab[5];          // si le tableau tab est publique
```

Parmi ces écritures, X=V[5] est la plus proche de la vision que l'on peut avoir du vecteur (quelque soit le type des éléments du vecteur). Les autres écritures sont moins simples et moins intuitives.
De plus, par exemple, dans le cas d'un vecteur de caractères, on peut prévoir le même opérateur qui permet de trouver l'indice d'un élément dont la valeur est égale à certain caractère.

L'opérateur []

```
#define N 50
class vect_char
{       int nb_elements;
        char tab[N];
  public :
        vect_char();                            // constructeur
        vect_char & operator+=(char C);         // ajout d'un caractère au vecteur
        const char operator[](int)              // pour rechercher par exemple V[5]
        int operator[](char);                   // pour connaître l'indice de V['f']
        .....
};
char vect_char::operator[](int X)               // renvoie V[X] s'il existe, 0 sinon
{if (X < 0 || X >= N)  {perror("mauvais indice"); return 0;}
  return tab[X];
}
int vect_char::operator[](char C)               // renvoie l'indice de V[C] s'il existe, -1 sinon
{for (int i=0; i < N; i++)
        if (tab[i] == C) return i;
  return -1;
}

void main()
{vect_char V;
  V = V + 'a'; ...
  printf("Le résultat de la recherche de l'élément d'indice 5 : %c \n", V[5]);
  printf("L'indice du caractère A dans le vecteur = %d \n", V['A']);
}
```

Remarque : le cas de *const char operator[](int);* est un cas d'utilisation de *const* à gauche d'une fonction.

Exercice et TP :

Ajouter les méthodes suivantes à la classe vecteur.
1- produit scalaire : $\sum(Ai \cdot Bi)$, A et B des vecteurs de même taille (contrôle)
2- produit vectoriel Ai Θ Bi : Ai est considéré comme une matrice d'une ligne et N colonnes, Bi comme une matrice de N lignes et d'une colonne (ou inversement). Par définition :

$a_1, a_2, \ldots, a_n \quad \Theta \quad \begin{matrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{matrix} \qquad a_1b_1, a_2b_2, \ldots, a_nb_n$

3- Somme de deux vecteurs : donne un vecteur d'éléments a_i+b_i
4- Multiplier les éléments d'un vecteur par une constante
Remarque : éviter la spécialisation en un vecteur d'entiers (prendre l'exemple d'un vecteur de string !).

Tableaux d'objets

Définition d'un tableau :

```
date t[10];                 // tableau de 10 dates
date *ptr = new date[20];   //tableau de 20 dates
```

Initialisation d'un tableau d'objets

Comme pour les tableaux ordinaires, on met les valeurs d'initialisation entre {}.

```
date t[2] = {date(1,"juin",1999), date(5,"mai" , 2000)};
date u[3] = {date(1,"juin",1999), date(5,"mai", 2000)};        // 2 sur 3 initialisées
```

Ce dernier cas ne fonctionne que s'il existe un constructeur sans argument pour la classe *date*.

Destruction d'un tableau d'objets
- Un tableau local d'objets est détruit à la sortie du bloc qui le déclare.
- Un tableau global ou statique d'objets est détruit à la fin du programme.
- Un tableau dynamique doit être libéré à l'aide de *delete[]*.

Entrées sorties en C++

Ecriture d'objet instance de classes de base

Il existe une classe prédéfinie **ostream** (flot de sortie) avec l'opérateur surchargé '<<' pour l'écriture de tous les types de base :

```
class ostream
{....
 public :
        ostream& operator<<(int);
        ostream& operator<<(long);
        ostream& operator<<(double);
        ostream& operator<<(float);
        ostream& operator<<(char);
        ostream& operator<<(const char *);
        ostream& put(char );
        .......
};
```

Il existe un objet **cout** prédéfini instance de la classe *ostream* qui permet les sorties à l'écran.

Exemples d'utilisation :

```
        int X=12;
        char C='z';
        char CH[] = "une chaîne de caractères";
        cout << X;
        cout << C;
        cout << "une chaîne de caractères" << CH << '\n';
```

L'opérateur '<<' est binaire, associatif à gauche et renvoie l'objet de la classe ostream. Ce qui permet de mettre des sorties en cascade comme dans : cout << X << Y << "chaîne " << ....

L'objet prédéfini ***cout*** désigne l'écran.
Pour pouvoir utiliser les entrées sorties, il faut inclure le fichier d'entêtes <iostream.h>.

Exemple :

```
#include <iostream.h>
void main()
{int X(10);         //i.e. X=10;  int est une classe de base prédéfinie avec ses constructeurs
 float F = 3.14;
 const char* CH = "une chaîne de caractères";
 cout << "X= " << X << "F= " << F << "une chaîne à la ligne \n" << CH << '\n';
}
```

L'intérêt de *cout* par rapport à *printf* est l'absence de chaîne de format (%d, %c, %s, ...)
cout interprète les caractères de contrôle (\n, \t, \" ...) comme dans *printf*.

Lecture des objets de classes de base

De même que pour les sorties, il existe une classe prédéfinie istream (flot d'entrées) avec l'opérateur '>>' surchargé pour lire tous les objets des classes de base :

```
class istream
{....
 public:
        istream & operator>>(int &);
        istream & operator>>(char &);
        istream & operator>>(char *);
        istream & operator>>(float &);
        istream & operator>>(double &);
        istream & operator>>(long &);
        istream & get(char &);
        istream & getline(char *, int);       // le 2e paramètre donne la taille maxi à lire, '\0'  compris
        istream get(char*, int , char='\n'); // le 3e paramètre = car de fin ('\n' par défaut)
        istream getline(char*, int );         //lecture d'une ligne
        .....
};
```

Il existe un objet **cin** prédéfini instance de la classe *istream* qui permet les sorties au clavier.
Exemples d'utilisation :

```
        int X; char C; char CH[15];
        cin >> X;
        cin << C;
        cout << "donner une chaîne de caractères";
        cin >> CH;        // problème si plus de caractères que 15, préférer  get/3
```

L'opérateur '>>' a les mêmes caractéristiques (symétriques) que celles de '<<'.

Exemples

Exemples d'utilisation de cin - cout

1- Lecture d'une série d'entiers (jusqu'à EOF) et rangement dans le tableau T :

```
int X, i=0;
int T[…];
while (cin >> X)              // terminer avec ctrl-d (ou ctrl-z sous windows)
        T[i++]=X;
```

2- Une autre méthode :

```
cin >> X;         // première valeur lue
while(! cin.eof())
        {T[i++]=X;
          cin >> X;
        }
```

3- Lecture d'au plus N entiers :

```
for(int i=0; i<N ;i++)
        {if (cin >> T[i])  continue;}
```

4- Lecture d'un tableau caractères jusqu'à '\n'

```
int i=0; char C;
while (cin.get(C))
      {T[i++]=C;
       if (C == '\n') break;
      }
```

5- Lecture d'un tableau d'au plus L lignes suivie de son écriture

```
const int N=256;
typedef char Ligne[N];
Ligne Tab[ L];
int n = 0;
while (n < L && cin.getline(Tab[n++], N));
n--;
for(int i=0; i<n; i++)
      cout << i+1 << Tab[i] << endl;
```

Remarque : le prédéfini *endl* est équivalent à un retour chariot.

Entrées Sorties des fichiers en C++

Quelques éléments de mise en forme :

- La fonction **width** permet de préciser la longueur minimum de caractères de la prochaine sortie <u>*numérique*</u> ou <u>*chaîne de caractères*</u> (mais pas les caractères) :

```
cout.width(5);
cout << '(' << 10 << ')' ;
```

donne : *(10)* "10" écrit sur 5 positions.

```
cout.width(4);
 cout << '(' << 10 << ')' << "ab" ;
```

donne : *(10)ab* "10" écrit sur 4 caractères; pas d'effet sur la chaîne "ab" qui est la deuxième sortie.

- **cout.width(0)** permet d'écrire sur autant de caractères que nécessaire.

- La fonction width permet de fixer le minimum que l'on peut déborder :

```
cout.width(4);
 cout << 123456;
```

donne : *123456* c'est à dire, on utilise le nombre de positions nécessaires. 4 positions est le minimum, mais on peut déborder.

<u>Remarque</u> : **width** n'affecte <u>que la prochaine sortie</u>.

- La fonction **fill** permet de préciser le caractère de "rembourrage / remplissage" de la prochaine sortie (cout).

```
cout.width(6);
cout.fill('#');
cout << '(' << "ECL" << ')' ;
```

donne : *(###ECL)* "ECL" écrit sur 6 caractères; zone remplie à gauche par des "#".

- La classe "**ios**" précise un certain nombre de valeurs par défaut pour le format des sorties. Ces valeurs sont modifiables à l'aide de la fonction "**cout.flags(X)**". La valeur de l'entier X sera une disjonction de valeurs constantes prédéfinies.

Pour ce faire, les triplets de bits (nombre octal) de drapeaux de format de sortie sont placés par un OU ('|') :

```
const int my_io_flags= ios::left | ios::showpoint | iso::fixed|ios::scientific|ios::unitbuf;
cout.flags(my_io_flags);
```

Où (les valeurs en octales) :

ios::left =02 :	remplissage après la valeur sortie
ios::showpoint=0400 :	affichage de '0' après la virgule
ios::fixed=010000 :	format virgule flottante fixe "dddd.dd"
ios::scientific=04000 :	format virgule flottant exposant ".ddddd E dd"
ios::unitbuf=020000 :	vidage de la sortie après chaque opération d'ES
ios::stdio=040000 :	vidage de la sortie après chaque caractère

On pourra sauvegarder les anciennes valeurs par :

```
int old_flags=cout.flags(new_flags);
// faire les sorties puis remettre les valeurs sauvegardées
cout.flags(old_flags);
```

Voir les autres valeurs possibles dans C++ - strustrup, p.344

- L'appel

```
my_stream.flags(my_stream.flags() | ios::showpos) ;
```

affecte l'affichage d'un '+' devant les nombres positifs sans modifier les autres flags.

- Pour réaliser cette même action, on peut utiliser **setf**

```
my_stream.setf(ios::showpos) ;    // fera exactement la même chose.
```

Les exemples suivants montrent les cas d'utilisation de setf.

Exemples : affichage de '1234' en différents formats

```
cout.setf(ios::oct, ios::basefield);  // octal
cout <<  1234;                // affiche 2322  (1234  en octal)

cout.setf(ios::dec, ios::basefield);  // décimal
cout <<  1234;                // affiche 1234 normalement   (décimal=défaut)

cout.setf(ios::hex, ios::basefield);  // octal
cout <<  1234;                // affiche 4d2 (1234  en hexadécimal)
```

- Pour afficher des nombres précédés de la base utilisée :

cout.setf(ios::showbase) à placer avant l'opération d'écriture.

Par exemple :

```
cout.setf(ios::showbase);
cout.setf(ios::oct, ios::basefield);  // octal
cout <<  1234;                  // affiche 02322  (0 au début : octal)

cout.setf(ios::hex, ios::basefield); // octal
cout <<  1234;                  // affiche àx4d2 hexadécimal (0x pour hexa)
```

Remarque : Il y aura rien pour la base décimal (seulement le nombre est affiché).

Remarque : la fonction **setf** est surchargée : **setf(long)** et **setf(long setbits, long field)**.

V. les sorties des flottants, Strustrup : p. 347

Entrées Sorties : fichiers et flots

Exemple de copie d'un fichier dans un autre. Le programme sera exécuté sous la forme :

$prog fin fout.

Il y aura donc 3 paramètres passés au programme qui doit copier *fin* dans *fout.*

```
#include <fstream.h>
#include <libc .h>

void erreur(char * s1; char * s2="") {
        cerr << s1 << ' ' << s2 << endl;
        exit(1);
}

int main(int argc, char * argv[]) {
        if  (argc != 3) error("nbr de param incorrect");

        ifstream source(argv[1]);               // argv[1] = nom de fin
        if (! source) erreur ("on ne peut pas ouvrier en lecture ", argv[1]);
        ofstream dest(argv[2]);                 // argv[2] = nom de fout
        if (! dest) erreur ("on ne peut pas ouvrier en écriture", argv[2]);

        char c;
        while (source.get( c))   dest.put( c);

        if (!source.eof() || dest.bad())                // sortie anormale de la boucle
                erreur("il y a eu un pb.");

        source.close(); dest.close();
        return(0);
}
```

Remarque : les fonction **eof** et **bad** sont définies dans la classe **ios**. Les états d'un flot sont :

int eof()	*fin de fichier*
int fail()	*la prochaine opération échouera*
int bad()	*le flot est altéré*
int good()	*la prochaine opération peut réussir*

Ces fonctions renvoient la valeur d'un bit défini dans *ios::io_state*. Ces valeurs constantes sont définies dans la classe *ios* :

Options d'ouverture : open_mode est un type énuméré avec les valeurs octales :

```
in = 1                  // ouvert en entrée
out = 2                 // ouvert en sortie
ate = 4                 // ouvert + aller à la fin
app = 010       // ajout (e.g. append)
trunc = 020             // tronque le fichier à une longueur 0
nocreate=040            // échoue si le fichier n'existe pas
noreplace=0100 // échoue si le fichier existe.
```

Exemple :

```
ofsteram dest(nom, ios::out | ios::nocreate);       // en écriture sans
if (ofstream.bad()) {                               //création (ne pas écraser)
        // ou dest.bad()
    …
}
```

On peut ouvrir un fichier en Entrée et en sortie :

```
fstream dico("mon_dico", ios::in | ios::out);
if (….)
```

Flot de chaîne de caractères

Exemple de flot de chaîne de sortie

```
char * ps = new char[taille];
ostrstream ost(ps, taille);         // flot de chaine en sortie de taille 'taille'
une_fonction(ost, …);       // taille non nécessaire. une_fonction remplit ost
display(ps);                        // écriture effective dans le fichier
```

Cette technique permet de remplir le tampon *ost* avant de l'écrire effectivement.

Exemple de flot de chaîne d'entrée

```
void ecr(char nom[], int t) {  //    afficher le contenu de "nom" de
    istrstream ist(nom, t);  //     taille t un mot par ligne
    char temp[MAX];
    while (ist >> temp)
            cout << temp << endl;
}
```

Remarque : l'initialisation de *cin*, *cout* et *cerr* sont faites dans la fonction prédéfinie *Io_init::Io_init()*.

Voir aussi *streambuff* dans Strustrup p. 357 pour la mise en tampon de données.

Surcharge de '<<' et de '>>' pour les objets définis par l'utilisateur

Il serait intéressant de pouvoir se servir des opérateurs '<<' et '>>' pour les entrées sorties des objets définis par l'utilisateur et harmoniser l'ensemble des écritures et des lectures.
Par exemple, on préfère écrire

```
complexe C;
cin >> C;          // lecture d'un nombre complexe
cout << "voici le nombre " << C;
```

Pour cela, il faut surcharger les opérateurs '<<' et '>>' dans les classes définies par l'utilisateur.

La classe *ostream* est une classe prédéfinie et fermée (non modifiable) en C++ . Il n'est donc pas possible de surcharger '<<' dans la classe *ostream* pour prendre en compte de nouveaux types. De même pour l'opérateur '>>'. La solution à ce problème est de déclarer une fonction amie dans la classe de l'utilisateur. **Ainsi, on donnera à ces deux opérateurs la permission d'accéder aux attributs de la nouvelle classe.**

Surcharge des opérateurs '<<' et '>>' : nombres complexes

```
class complexe {
        double re, im;
public :
        ...
        friend ostream& operator<<(ostream &, const complexe &);
        friend istream& operator>>(istream &, complexe &);
        ....
};
ostream & operator<<(ostream &f, const complexe & C) {
         f << "(" << C.re << "," << C.im << ")" << endl;
         return f;
}
istream & operator>>(istream &f, complexe & C) {
        f  >> C.re  >> C.im ;  return f;
}
void main() {
        complexe X(1.5, 2.8), Y ;
        cout << X;                          // produit à l'écran (1.5, 2.8)
        cin >> Y; cout << Y;          // lecture et écriture d'un complexe
}
```

Pour toutes les classes, on peut se servir du format de définition ci-dessus.
Il ne faut pas oublier de renvoyer le flot de type *ostream* afin de pouvoir mettre les écritures en cascade et de mélanger différentes classes. De même pour *istream*.
Le format de "<<" s'impose car c'est un opérateur binaire; la classe *ostream* est fermée et ne prévoit pas l'écriture d'un nombre *complexe*. Son 1er paramètre étant de type *ostream*, il ne peut être qu'ami. Son 2e opérande doit être un nombre complexe (règle de surcharge d'un opérateur), il ne peut pas être externe. Il pourrait ne rien renvoyer (*void*) si on ne veut pas écrire en cascade et mixter.

Remarque : il existe un objet prédéfini **cerr** du même comportement que 'cout' réservé pour les sorties sur le flot d'erreurs. 'cerr' est par défaut redirigé vers 'cout'.

Exercice : compléter la classe *vecteur* en ajoutant les opérateurs '[]', '<<' et '>>' . Prévoir l'agrandissement du vecteur en cas d'ajouts successifs (l'opération ajout devient totale).

Attributs constants et statiques

Les attributs d'une classe peuvent être
- de type de base
- un type construit
- un objet d'autre classe (voir plus loin)

Comme nous l'avons déjà vu, l'attribut d'un classe est une **variable d'instance**. Cet attribut figure dans chaque instance de la classe mais il est spécifique à cette instance.

Un attribut de classe peut être **statique**. Dans ce cas, cet attribut appelé **variable de classe** est commun à toutes les instances de la classe. Tous les objets de la classe se partagent cet attribut. Ceci permet de mettre en place des compteurs d'instance ou d'autres contrôles sur les créations d'objets.

Exemple de données statique (variable de classe)

```
class X{
        static int S;
        static const int size =  5;
public :
        char array[size];
        void add();
        void affiche();
};
void X::add()     {  S++ ;}          // compteur des créations incrémenté

void X::affiche()
        {printf("la valeur de S =  %d \n", S); }

int X::S=0;                 // initialisation hors toute fonction

void main()
{X A;
 A.add();
 A.affiche();
 X B;
 B.add();
 B.affiche();
}
```

La déclaration d'une variable statique dans la classe ne réserve pas de place pour cette variable. C'est pourquoi dans l'exemple ci-dessus, la fonction add déclare la variable statique x (cf. manuel C++).

Une classe peut également déclarer un attribut constant. Cette constante ne sera pas modifiable.
Dans l'exemple ci-dessous, on remarque la méthode d'initialisation de l'attribut constant.

```
class truc
{const int x;
  const int taille = 100;                  // initialisation à la déclaration
  float y;
  public :
        truc(int i, float j) : x(i)        // x doit être initialisé avec cette méthode
                {y = j;}
};
```

On peut remarquer qu'une autre manière d'initialiser une constante est pendant sa déclaration.

Remarque : certains programmeurs préfèrent utiliser un type *enum* pour déclarer une constante :

```
class thing
{enum {taille=100};
  int table[taille];
  public :
        .....
};
```

Initialisation des attributs d'une classe

Dans les exemples que nous avons vus, les attributs sont initialisés dans les constructeurs par des opérations d'affectation classiques.
Une autre manière plus simple d'initialiser ces attributs est montrée dans l'exemple suivant :

```
class thing
{int x;
 char c;
 public :
        thing(int i, char j) : x(i) ,  c(j)          // constructeur
        { }
        ...
};
```

Cette méthode d'initialisation des attributs n'est utilisable que dans les constructeurs de la classe.

Un exemple complet : classe String

La manipulation des chaînes de caractères en C/C++ passe par les fonctions de la bibliothèques dont les prototypes sont définis dans "string.h".
On peut réaliser une classe String avec les opérateurs classiques pour faciliter la manipulation des chaînes de caractères.
On pourra utiliser cette classe en écrivant par exemple:

```
String S1 = "Oxford";                  // initialisation par constructeur
String S; S2="université";             // copie constructeur
String S3 = S1 + S2;                   // concaténation de chaînes
S2 += 'a';                             // ajout d'un caractère;
S2 = 'x' + S2;                         // un autre type d'ajout
char X = S1[5];                        // accès à un caractère
int i = S2['s'];                       // indice de 's' dans S2
S2 = S1;                               // affectation
if (S1 == S3 || S2 != S1 && S1 > S2) ..          // tests
cout << S1;                                 // écriture à l'écran
cin >> S2;                             // lecture au clavier
.....
```

Remarques :

== : string x string bool
== : string x char* bool
== : char* x string bool
== : char* x char* bool (e.g. if ((string)"toto" == "titi") …)

Remarques :

- dans le 1e et 2e cas, "==" peut être une fonction membre string x string.
- dans le 3e cas, "==" doit être déclaré ami
- dans le dernier cas, sans la conversion, l'opérateur '==' de char* s'applique.

Exemple complet string

Fichier String.hpp

```
#ifndef _STRING_HPP
#define _STRING_HPP

#include <iostream.h>

class String
{       int size;
        char *chaine;
 public :
        String();                                       // constructeur vide
        String(const char*s);                           // constructeur avec char *
        String(const String & s);                       // copie constructeur

// En général, le & a gauchede '='  n'est pas nécessaire sauf si l'on
 // compte écrire (e.g. (S=S3)="toto") qui est sans sens.
        String& operator=(const String & s);            // affectation

        int operator ==(const String& s)const;          // test d'égalité
        int operator !=(const String& s) const;         // inégalité

        friend String operator+(const String &s) const ;       // String + char* et String + String
        friend String operator+(const char c) const;           // String + char
        friend String operator+(const char c, const String& s) ;    // char + String

        char& operator[](int i);                        // le ième caractère
        ~String();                                      // destructeur
        char * texte() const;                           // accès au contenu
        friend ostream& operator<<(ostream& f, const String &S);       // sortie
        friend istream& operator>>(istream& f, String &S);                  // entrée d'une ligne
        bool substr(String s);                          // s est sous chaîne de *this
        void edit(ostream& f);                          // édition sur un flot quelconque
 };

#endif
```

Fichier String.cpp

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

String::String()
        {chaine=new char[1]; chaine[0]=0; size=0;}

 String::String(const char*s) {
        size=strlen(s);chaine=new char[size+1];
        strncpy(chaine,s, size+1);
         }

 String::String(const String & s){
        size=s.size;chaine=new char[size+1];
        strncpy(chaine,s.chaine,size +1);
}

 String String::operator=(const String & s){
        if (this == &s) return *this;
        this->~String(); // ou delete[] chaine;
        size=s.size; chaine=new char[size+1];
        strncpy(chaine,s.chaine,size+1 );
        return *this;
}

// Friend pour pouvoir utiliser : (string x string), (String x char*) et (char* x String).
// Pour (char* x char*), utiliser un cast.
// Une fonction AMIE ne peut pas être const.

bool operator ==(const String &s1, const String & s2)
        {return strcmp(s1.chaine,s2.chaine)==0;}

 bool operator !=(const String & S1, const String & S2)

 { return (strcmp(S1.chaine , S2.chaine) != 0);}

// Les memes raisons que '=='
 String operator+(const String &s1, const String & s2)
   {String s3; s3.size=s1.size+s2.size;
    s3.chaine=new char[s3.size+1];
    strncpy(s3.chaine, s1.chaine, s1.size+1);
    strncat(s3.chaine,s2.chaine,s2.size+1);
    return s3;}

 String operator+(const String& s,const char c) {
    char  ch[]={c,0}; return s+ch;    //conversion ch en String
}
```

```
// ATTENTION : le '+' des chaines n'est pas commutative !!
String operator+(const char c, const String& s) {
   char ch[]={c,'\0'};    //conversion ch en String
   return ch+s;
}

char& String::operator[](int i)    // i de 0 a N-1, le '&' pour lvalue
   {assert ((i>=0) && (i<size)); return chaine[i];}

String::~String()
       {delete[] chaine;
        chaine=NULL; size=0;        // important
       }

char * String::texte() const {return chaine;}

ostream& operator<<(ostream& f, const String &S)
  {return f<<S.chaine;}

istream& operator>>(istream& f, String &S)
  {char ch[256];
   f.getline(ch,255);     // lecture d'une ligne
   S.size=strlen(ch);S.chaine=new char[S.size+1];
   strncpy(S.chaine,ch, S.size+1);
   return f;}

bool dans(char* ch1, char* ch2);
bool String::substr(String s) // s est sous chaîne de *this
       {return dans(s.chaine,chaine);}

// une des solution de recherche de CH1 sous chaîne de CH2
bool dans(char ch1[], char ch2[])
{int i1,i2,l1,l2;
 l1=strlen(ch1); l2=strlen(ch2);
 i1=i2=0;
 while (i1<l1 && i2<l2)      // et l2-l1 ?
       if (ch1[i1++] != ch2[i2++])
              {
              if (i1!=1) {i2--; i1=0;} // eg : "abc" et "ddababcff"
               else i1=0;
              if (i2 > l2-l1)  break;
              }
 if (i1==l1) return true;
 return false;
}

 void String::edit(ostream& f)       // une autre façon d'afficher
 {f << "la chaine : " << chaine;
 }
```

Test de la classe String :

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

int main()
{String S="toto"; cout << S << endl;
 String S1 = "titi";cout << S1 << endl;
 if (S == S1) cout << "S == S1 \n"; else cout << "S != S1 \n";
 if (S == "toto") cout << "S == toto \n"; else cout << "S != toto \n";
 if ("toto" == S) cout << "toto==S \n"; else cout << "toto!=S \n";
 if (String("toto") == "toto")
       cout << "String(toto)==toto \n";
else cout << "(String)toto!=toto \n";

 cout << S << endl;
if (S != "toto") cout << "S != toto\n" ;
 else cout << "S == toto\n";

 if (S != S) cout << "S != S \n" ;
 else cout << "S == S \n";

 if (S != S1) cout << "S != S1 \n" ;
 else cout << "S == S1 \n";

 cout << S << " + " << S1 << " donne : " << S+S1 << endl;
 cout << S << " + " << "tata" << " donne : " << S+"tata" << endl;
 cout << "trtr" << " + " << S << " donne : " << "trtr" +S << endl;
 cout << "trtr" << " + " << "tptp" << " donne : ";
       cout << (String)"trtr" + "tptp" << endl;

 cout << S << " + " << 'Z' << " donne : " << S+'Z' << endl;
 cout << 'O' << " + " << S << " plus 'Z' donne : " << 'O'+S+'Z' << endl;

 String S3,S4;
 S4=S3="blabla";
 //S3="blabla";S4="blabla";
cout << "S3= " << S3 << " et S4=" << S4 << endl;

 cout << "S= " << S << endl;
 if ((S3 = S) !=  "toto") cout << "pb \n" ;
 else cout << "ok\n";

 // Ici, si '=' ne renvoie pas '&', on aura S=S3=toto
 // si '=' renvoie '&', on aura S=toto et S3=blibli. ( Normal).
 cout << "S= " << S << endl;
 (S3 = S) =  "blibli" ;  // test de '=' avec ou sans '&' au retour.
 cout << "Apres affectation a gauche : " ;
 cout << "S= " << S << endl;
 cout << "S3= " << S3 << endl;
```

```
S="Centrale"; S1="Ecole Centrale de Lyon";
 if (S1.substr(S)) cout << S << " est sous chaine de " << S1 << endl;
 else cout << "pb de substr \n";

return 0;
}
```

Trace d'exécution :

```
toto
titi
S != S1
S == toto
toto==S
String(toto)==toto
toto
S == toto
S == S
S != S1
toto + titi donne : totititi
toto + tata donne : totototata
trtr + toto donne : trtrtoto
trtr + tptp donne : trtrtptp
toto + Z donne : totoZ
O + toto plus 'Z' donne : OtotoZ
S3= blabla et S4=blabla
S= toto
ok
S= toto
Apres affectation a gauche :
S= toto
S3= toto
Centrale est sous chaîne de Ecole Centrale de Lyon
```

Exercice : compléter la classe *String* et ajouter les opérateurs qui manquent (voir l'énoncé).

Réutilisation par composition de classes

Un membre d'une classe peut être :
- une fonction (méthode)
- une donnée d'un type de base ou construite
- un objet instance d'une autre classe

La présence d'un objet d'une autre classe dans une classe produit une **composition**.
Une classe A avec un membre instance d'une classe B définit une relation "**partie de**" entre B et A.

Exemple :
On suppose que les classes *Date* et *String* existent.

```
class vin{
        String Nom;             // utilisation de la classe String
         String Région;
         char * Sépage;         // on pourrait encore utiliser String
         int Numéro;
         Date Millésime;        // utilisation de la classe Date
public :
  ....
};
```

Règles sur les constructeurs et destructeur d'une classe composée

- Le constructeur de la classe Composée (agrégat) doit appeler les constructeurs des classes qui la composent **avant** d'effectuer sa propre initialisation.

- La règle d'appel du constructeur vide : dans le cas de la composition de classes, le constructeur vide de la classe Composée (Agrégat) appellera le constructeur vide des composantes.

- Les appels aux constructeurs des membres Composant sont introduits par **':'** et sont situés entre l'entête et le corps du constructeur de la classe Agrégat.

- Les **destructeurs** des membres composants sont appelés **automatiquement** par le destructeur de la classe Composée. Le destructeur de la classe Agrégat n'a donc pas besoin d'appeler les destructeurs des classes Composants.

- Les classes de base (int, char, etc.) sont déjà définies et disposent de constructeurs par défaut. Ce qui permet de les initialiser comme les attributs instances d'autres classes.

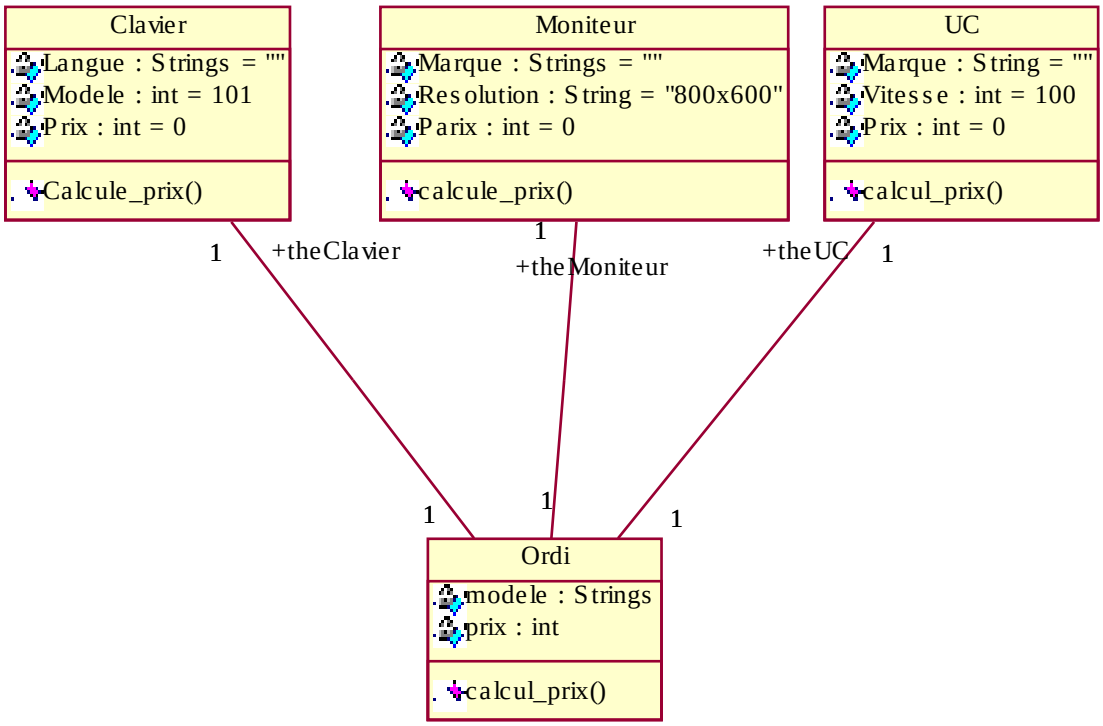
- Une fonction membre de la classe Agrégat peut appeler les fonctions publiques membres d'une classe Composant par l'intermédiaire du membre Composant (par exemple **U.ecrit()** de l'exemple suivant).

Exemple de composition de classes (agrégation)

```
class Composant
{int X ;
 public :
        Composant(int y) : X(y) {}
        void ecrit() const {printf("%d\n", X); }
};

class Agrégat
{float Z;
 Composant U;          // composition à l'aide des instances de Composant
 Composant V;
public :
 Agrégat (float, int , int);          // constructeur
 void ecrit() const;
};

Agrégat:: Agrégat (float a1, int a2, int a3) : U(a2), V(a3)
        { Z = a1 ;}

void Agrégat::ecrit() const
{printf("%g \n", Z);
 U.ecrit();            // appel d'une fonction de la classe Composant
 V.ecrit();
}
```

N.B. lorsque une instance de la classe Agrégat contient plusieurs composantes, l'agrégation peut être réalisée par une contenance physique (voir aussi la partie UML, 2e partie de ce document).

Un exemple complet : ordinateur



Un ordinateur est composé d'un clavier, un moniteur et une unité centrale. Nous verrons plus loin le formalisme UML et les détails de son utilisation.

```
#include <iostream>
#include <string>
using namespace std ;

char* T_langue[]={"Fr", "Tr", "Us", "Gb", "Du"};

class unite_centrale { … };

class clavier {
        int nb_touches;
        string langue;      // Fr, us, ...
public:
        clavier(int nb=101, string l= "Fr") {          // contrôler la langue 'l'  par T_langue
                nb_touches=nb, langue=l;
         }
        void edit() {
                cout << "clavier : " << nb_touches << "touches ";
                 cout << " Langue : " << langue << endl;
        }
};

class moniteur {
        int res_h, res_v;
        int taille;          // 14, 15, 17 ... ...

public:
        moniteur(int rh=1024, int rv =768 , int t=17) : res_h(rh), res_v(rv), taille(t) {}
        moniteur(const moniteur & m) : res_h(m.res_h), res_v(m.res_v), taille(m.taille) {}
        void edit() {cout << "Moniteur : res " << res_h << "x" << res_v;
                        cout << " Taille : " << taille << endl;
        }
};

class ordinateur {
        clavier * p_clavier;
        moniteur * p_moniteur;
        unite_centrale *p_uc;
        int prix;
        string marque;
public:
        ordinateur() : p_clavier(NULL), p_moniteur(NULL), unite_centrale(NULL) {
                prix = 0, marque = "";
        }
        ordinateur( clavier & cl,  moniteur & m, unite_centrale& u ) {
                p_clavier=new clavier(cl);
                p_moniteur=new moniteur(m);      // appel du copie constructeur de "moniteur"
                p_uc=new unite_centrale (u);     // appel du copie constructeur de "unite_centrale"
        }
        ~ordinateur() { … }
```

```
        void edit() {
                cout << "L'ordinateur : marque "<< marque << " prix " << prix ;
                if (p_clavier) p_clavier->edit();
                if (p_moniteur) p_moniteur->edit();
                if (p_uc) p_uc->edit();
        }
};

int main() {
        clavier cl(102, "Fr");
        moniteur m(800,600,15); unite_centrale u(…);
        ordinateur o1(cl, m, u);
        o1.edit();
        return 0;
}
```

Remarque : on pourra déclarer *ordinateur* ami de ses composantes pour lui donner accès à la partie privée de ces classes. On peut donner cet accès seulement à une des fonctions de "ordinateur".

Classes imbriquées

- Une classe imbriquée est une classe déclarée à l'intérieur d'une autre classe.
- La classe imbriquée est locale à la classe englobante.
- Une classe imbriquée peut être dans la zone public ou privée de la classe englobante.

Exemple de classe imbriquée publique

```
class extérieure
{ int Ve;
  public :
        class intérieure
        { public :
                int Vi;
                intérieure(int i) {Vi = i;}             // constructeur
                void affiche() {cout << Vi; }
        };
        extérieure(int i) {Ve = i; }                    // constructeur
        void f()                                        // fonction membre d'extérieure
                {intérieure x(2); x.affiche();}             // Appelle une fonction membre d'intérieure
};

void main()
{extérieure ex(10);
 extérieure::intérieure in(5);
 ex.f();
 in.affiche();              // accès à la fonction membre de la classe intérieure
}
```

Remarque : dans l'exemple précédent, même si la classe *intérieure* est privée, C++ (version ?) donne encore accès à :
extérieure::intérieure in(5) alors que normalement, intérieure est invisible à l'extérieur.

Définition des méthodes des classes imbriquées

```
class ext
{int Ve;
  public :
        class inter1
              {public :
                    int V1;
              };
        class inter2
              {public :
                    int V2;
              };
   inter1 f(inter2);       // fonction membre de 'ext', accepte un paramètre 'inter2' et renvoie
'inter1'
};

ext::inter1 ext::f(ext::inter2 X)      // définition de la fonction membre 'f'. Elle accepte un
      { .... }                         // paramètre de type 'inter2' et renvoie une valeur de type 'inter1'

void main()
{ext ve;
  ext::inter1 vi1;
  ext::inter2  vi2;
  vi1 = ve.f(vi2);                     // appel de la fonction 'f' de 'ext'
}
```

Un exemple complet : une liste chaînée de String

La classe **Liste** de String est composée d'une classe interne **Boite**.
Pour faciliter les insertions, on maintient un pointeur sur la dernière boite (attribut fin).
Quelques opérateurs sont définis. Il faudra les compléter.

Classe liste de string
Fichier Liste.hpp

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

class Liste
{
      class Boite                  // on peut préférer une structure (struct) et y mettre,
            {String info;          // des constructeurs  des zones private, etc.
             Boite * svt;
             public :
             Boite();
             Boite(const String, Boite *);
             Boite(const Boite&);
             Boite& operator=(const Boite&);
             String & get_info() {return info;}      // accès à info (privé) Inline
             Boite* & get_svt() {return svt;}        // accès à svt (privé) Inline
             void edit();
            };
 Boite * ancre;          // pointeur de tète (1ère boîte)
 Boite * fin;            // pointeur de fin (dernière boîte)
 public :
 Liste();
 Liste(const Liste &);
 Liste & operator=(const Liste &);
 Liste operator+(const String);          // ajout d'un élément à la fin
 ~Liste();
 void insere_fin(String);
 void edit();
};
```

Fichier Liste.cpp

```
// définitions pour la classe Boite
Liste::Boite::Boite() {}                                    // constructeur vide
Liste::Boite::Boite(const String I, Boite *S=NULL)          // constructeur
      {info=I; svt=S;}
Liste::Boite::Boite(const Boite & B)                  // copie constructeur
      {info=B.info; svt=B.svt;}
void Liste::Boite::edit() {cout << info <<" ";}             // édition
Liste::Boite& Liste::Boite::operator=(const Boite & B)  // opérateur d'affectation
      {info=B.info; svt=B.svt;
       return *this;
      }
// les fonctions de la classe Liste
Liste::Liste() {ancre=fin=NULL;}
```

```
Liste::Liste(const Liste & L)
{Boite * B=L.ancre;
 ancre=NULL;
 while (B != NULL)
        {insere_fin(B->get_info());          // info est privé
         B=B->get_svt();}                    // svt est privé
}
void Liste::insere_fin(String S)
{if (ancre == NULL)
        {ancre=new Boite(S);fin=ancre; return;}
 fin->get_svt()=new Boite(S);
 fin=fin->get_svt();
}
void Liste::edit()
{Boite * B=ancre;
 while(B != NULL) {B->edit(); B=B->get_svt();}
 cout << endl;
}
Liste::~Liste()
{Boite * B=ancre, *C;
 while (B != NULL)
        {C=B; B=B->get_svt();delete C;}
 ancre=fin=NULL;
}
Liste & Liste::operator=(const Liste & L)
{Boite * B=L.ancre;
 ancre=NULL;
 while (B != NULL)
        {insere_fin(B->get_info());          // info est privé
         B=B->get_svt();}                    // svt est privé
 return *this;
}
Liste Liste::operator+(const String S)
{Liste L=*this; L.insere_fin(S);
 return L;
}
```

Fichier Appli.cpp

```
void main()
{Liste L;
 L.insere_fin("toto"); L.edit();
 L.insere_fin("titi"); L.edit();
 Liste L1=L; L1.edit();
 Liste L2; L2=L1; L2.edit();
 Liste L3=L2 + "blabla"; L3.edit();
}
```

Exercice : lire une phrase (suite de mots avec cin) et compter le nombre d'occurrence de chaque mot.