LE LANGAGE C++

SAFSOUF Yassine

Docteur en informatique et chef du département informatique
Responsable des programmes pour les filières d'informatique

Table des matières

LE LANGAGE C++

Éléments préalables

Structure d'un programme C++

La structure minimale d'un programme C++ est similaire à celle d'un programme C. Elle peut ensuite être étendue par des éléments (fonctions, instructions, structures de contrôle, etc...) abordés lors de l'étude du langage C, et par les éléments propres au C++.

La fonction principale, appelée lors du début de l'exécution d'un programme, est la fonction **main**

```
int main() { ... } // Programme sans paramètre

/* Programme avec paramètres :
   argc : nombre de paramètres (y compris le nom du programme)
   argv : tableau de paramètres, argc entrées
*/
```

Les Fonctions

Les fonctions C++ se déclarent et se définissent comme en C. Plusieurs caractéristiques ont cependant été ajoutées ou modifiées :

- **Possibilité de définir des valeurs par défaut pour certains paramètres de fonctions.**
- **Définition de fonctions inline.**
- **Surcharge des noms de fonctions**

Les Fonctions (valeurs par défaut)

Certaines fonctions sont appelées avec des paramètres qui changent rarement. Il s'agit de la définition de valeurs par défaut fournies lors de la déclaration de la fonction.

Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut. Exemple :

```
int calculer (int a=0, int b= 0);
```

Les Fonctions (valeurs par défaut)

```
int calculer (int a=0, int b= 0);
```

Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants peuvent alors être omis (ainsi que les virgules correspondantes), les paramètres formels seront initialisés avec les valeurs par défaut. Exemples :

```
c= calculer (5,8);        // a = 5, b = 8
c= calculer (2);          // a = 2, b = 0
c= calculer ();           // a = 0, b = 0
```

Les Fonctions (inline)

Normalement, un appel de fonction est une *rupture de séquence* : à l'endroit où un appel figure, la machine cesse d'exécuter séquentiellement les instructions en cours ; les arguments de l'appel sont disposés sur la pile d'exécution, et l'exécution continue ailleurs, là où se trouve le code de la fonction.

Les Fonctions (inline)

Une fonction en ligne est le contraire de cela : là où l'appel d'une telle fonction apparaît il n'y a pas de rupture de séquence. Au lieu de cela, le compilateur remplace l'appel de la fonction par le corps de celle-ci, en mettant les arguments effectifs à la place des arguments formels.

Les Fonctions (inline)

Cela se fait dans un but d'efficacité : puisqu'il n'y a pas d'appel, pas de préparation des arguments, pas de rupture de séquence, pas de retour, etc. Mais il est clair qu'un tel traitement ne peut convenir qu'à des fonctions **fréquemment appelées** (si une fonction ne sert pas souvent, à quoi bon la rendre plus rapide ?) de **petite taille** (sinon le code compilé risque de devenir démesurément volumineux) et **rapides** (si une fonction effectue une opération lente, le gain de temps obtenu en supprimant l'appel est négligeable).

Les Fonctions (inline)

En C++ on indique qu'une fonction doit être traitée en ligne en faisant précéder sa déclaration par le mot **inline** :

```
inline int abs(int y)
{
      return (y >= 0) ? y : -y;
}
```

```
void main()
{
      int x = -3;
      x = abs(x);    //  x = (x >=0) ? x : -x;
}
```

Surcharge des noms de fonctions

La *signature d'une fonction* est la suite des types de ses arguments formels (et quelques éléments supplémentaires, que nous verrons plus loin).

Le type du résultat rendu par la fonction ne fait pas partie de sa signature.

Surcharge des noms de fonctions

en C++ des fonctions différentes peuvent avoir le même nom, à la condition que leurs signatures soient assez différentes pour que, lors de chaque appel, le nombre et les types des arguments effectifs permettent de choisir sans ambiguïté la fonction à appeler. Exemple :

int puissance(int x, int n) **{ calcul de xn avec x et n entiers }** **double puissance(double x, int n)** **{ calcul de xn avec x flottant et n entier }**	**double puissance(double x, double y)** **{ calcul de xy avec x et y flottants }**

Surcharge des noms de fonctions

On voit sur cet exemple l'intérêt de la surcharge des noms des fonctions : la même notion abstraite « x à la puissance n » se traduit par des algorithmes très différents.

Or le programmeur n'aura qu'un nom à connaitre, **puissance**. Il écrira dans tous les cas

```
c = puissance(a, b);
```

le compilateur se chargeant de choisir la fonction la plus adaptée, selon les types de a et b.

Premier programme

① ②

```
#include <iostream>
using namespace std;
main ()
{
  int a;
  cout << "Entrez la valeur de a: ";
  cin >> a;
  cout << " a = " << a << endl;
  cin.get();
  cin.get();
}
```

③

Entrées-sorties simples

Cette section traite de l'utilisation simple des flux standard d'entrée-sortie, c'est-à-dire la manière de faire en C++ les opérations qu'on fait habituellement en C avec les fonctions **printf** et **scanf**.

Un programme qui utilise les flux standard d'entrée-sortie doit comporter la directive

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

ou bien, si vous utilisez un compilateur récent et que vous suivez de près les recommandations de la norme 4 :

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

Entrées-sorties simples

Les flux d'entrée-sortie sont représentés dans les programmes par les trois variables :

- **cin** : Représente l'entrée standard. En général, cin permet de lire les données depuis le terminal de l'ordinateur.
- **cout** : Représente la sortie standard.
- **cerr** : Représente les erreurs standard. Là aussi, c'est généralement l'écran qui est utilisé, et la syntaxe est identique à **cout**.

Entrées-sorties simples (cin)

Exemple :

```
cin >> x ; // Clavier ==> Variable x
```

L'opérateur de redirection « **>>** » sert à assurer le transfert de l'information issue du clavier de l'ordinateur vers la variable x. Le sens de lecture va donc de la gauche vers la droite. La variable x peut-être de n'importe quel type prédéfini, par contre, il n'est pas possible de placer à cet endroit une constante littérale.

Entrées-sorties simples (cout)

Exemple :

```
cout << x ; // Ecran <== Variable
```

L'opérateur de redirection « **<<** » sert à assurer le transfert de l'information issue de la variable **x** vers l'écran de l'ordinateur. Le sens de lecture va donc de la droite vers la gauche. La variable **x** peut-être de n'importe quel type prédéfini. Il est possible d'enchaîner plusieurs opérateurs de redirections.

```
cout << " La valeur de x est :  " << x ;
```

Entrées-sorties simples (cerr)

Exemple :

```
cerr << « erreur de transmission » ;
```

le flux standard pour la sortie des messages d'erreur (l'équivalent du **stderr** de C), également associé à l'écran du poste de travail.

Entrées-sorties simples

Les écritures et lectures sur ces unités ne se font pas en appelant des fonctions, mais à l'aide des opérateurs **<<**, appelé ***opérateur d'injection*** (« injection » de données dans un flux de sortie), et **>>**, appelé ***opérateur d'extraction*** (« extraction » de données d'un flux d'entrée).

Travaux Pratique (TP 1)

Écrire un programme qui calcule le prix TTC tel que :

- La fonction "CalculeTTC" comportera deux paramètres formels, le premier est le prix HT et le deuxième est la TVA (par défaut initialisé à 14%). Cette fonction doit être écrite cette façon :
  - **float CalculeTTC (……)**

Une fois que le programme est lancé, l'utilisateur devra saisir une valeur quelconque sans les taxes. Le programme doit afficher par la suit la valeur TTC.

Travaux Pratique (TP 2)

Ce programme propose d'élaborer une facture par rapport à trois articles prédéterminés. Cette fois-ci, l'utilisateur doit saisir la valeur en dirhams et hors taxe de chacun des articles. Le programme recense chacune de ces valeurs et en profite pour donner la valeur de l'article avec les taxes.

Travaux Pratique (TP 2)

```
Quel est votre prix hors Taxe en Dirhames de la chemise ? 220
Quel est votre prix hors Taxe en Dirhames du pantalon ? 300
Quel est votre prix hors Taxe en Dirhames des chaussures ? 369

|------------|---------|----------|
| Article    | Prix HT | Prix TTC |
|------------|---------|----------|
| Chemise    |  220.00 |   250.80 |
| Pantalon   |  300.00 |   342.00 |
| Chaussures |  369.00 |   420.66 |
|------------|---------|----------|
| Totaux     |  889.00 |  1013.46 |
|------------|---------|----------|
```

Formatage de l'information

Il est possible de formater l'information qui est envoyé à l'écran plutôt que d'avoir la présentation par défaut. On utilise alors des manipulateurs qui sont des opérateurs prédéfinis et qui modifient la valeur envoyée (avant l'affichage) pour que sa présentation soit plus adaptée. La syntaxe est relativement simple, il suffit de placer le manipulateur à la suite de l'opérateur de redirection :

```
Flot << manipulateur // pour un flot de sortie
Flot >> manipulateur // pour un flot d'entrée
```

Formatage de l'information

Avec ce type de manipulateur, vous remarquez que nous n'avons pas besoin de repréciser le même traitement pour les variables qui suivent. Une fois que l'on place un manipulateur, il reste actif. Si vous désirez réobtenir le comportement par défaut, vous devez appliquez le manipulateur adéquat.

Un programme qui utilise les instructions de formatage doit comporter la directive **#include <iomanip.h>**

Formatage de l'information

- **setw (nombre)** : Définit le gabarit de la variable à afficher avec une justification à droite par défaut. Si la valeur à afficher est plus importante que le gabarit, cette valeur ne sera pas tronquée et sera donc affichée de façon conventionnelle. **Le manipulateur « setw » doit être utilisé pour chacune des informations à afficher.**
- **setfill (caractère)** : Définit le caractère de remplissage lorsqu'on utilise un affichage avec la gestion de gabarit. Par défaut, le caractère de remplissage est l'espace.

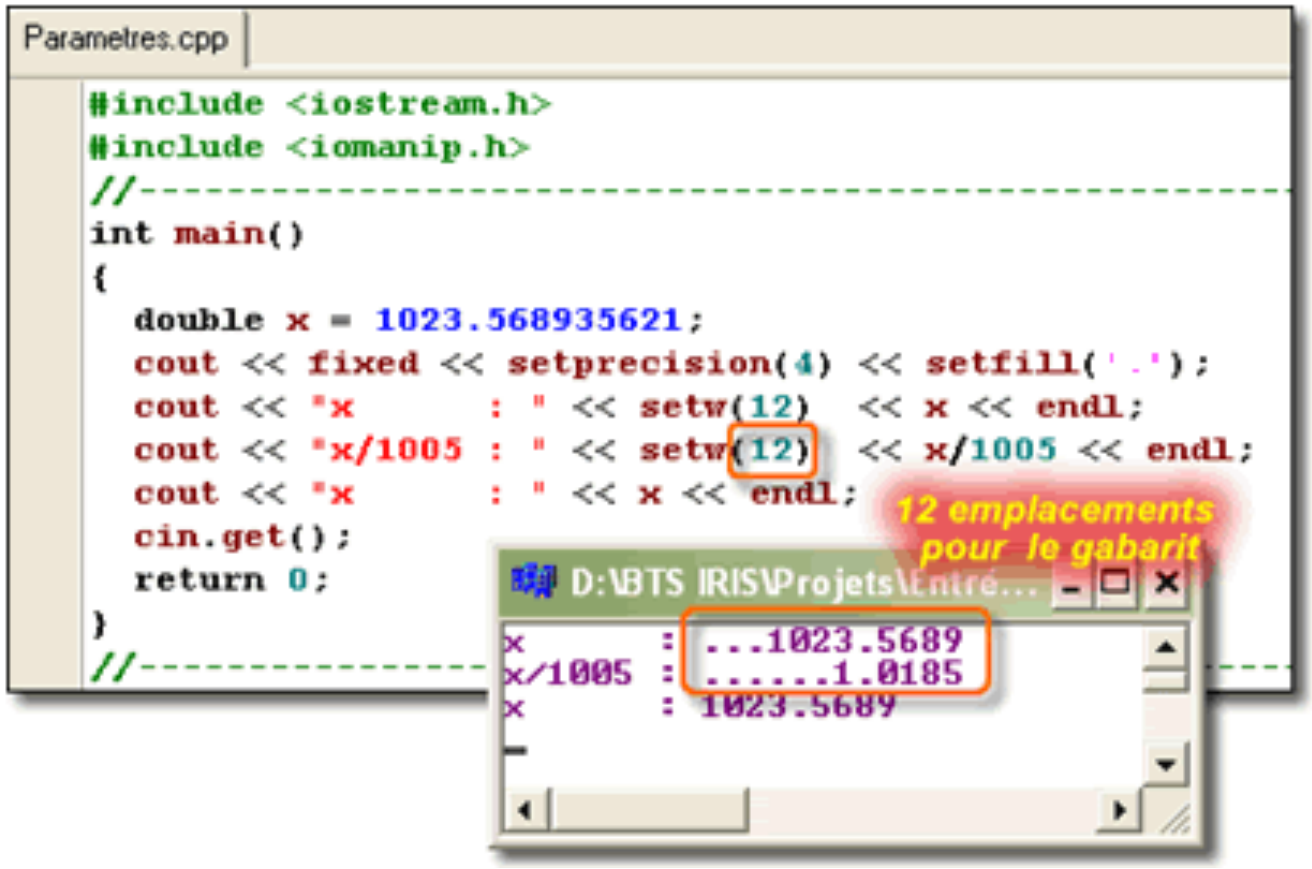
Formatage de l'information

- **setprecision (nombre)** : Permet de définir le nombre de chiffres significatifs pour les nombres réels. C'est uniquement pour l'affichage, la variable garde sa précision, et la valeur affichée est arrondie. Lorsque l'on utilise au préalable le manipulateur « **fixed** », le manipulateur « **setprecision** » permet d'indiquer le nombre de chiffres significatifs après la virgule.
- **fixed** : Notation « point fixe » pour les nombres réels : 10.569
- **scientific**: Notation scientifique des nombres réels : 1.569 e+01

Formatage de l'information

En ce qui concerne « **setw** », sachez que ce manipulateur définit uniquement le gabarit de la **prochaine information** à écrire. Si l'on ne fait pas de nouveau appel à « **setw** » pour les informations suivantes, celles-ci seront écrites suivant les conventions habituelles, à savoir en utilisant l'emplacement minimal nécessaire pour les écrire.

Formatage de l'information



Allocation dynamique de mémoire

Des différences entre C et C++ existent aussi au niveau de l'allocation et de la restitution dynamique de mémoire.

Les fonctions **malloc** et **free** de la bibliothèque standard C sont disponibles en C++. Mais il est fortement conseillé de leur préférer les opérateurs **new** et **delete**. La raison principale est la suivante : les objets créés à l'aide de new sont initialisés à l'aide des constructeurs correspondants, ce que ne fait pas malloc.

Allocation dynamique de mémoire

De même, les objets liberés en utilisant delete sont finalisés en utilisant le destructeur de la classe correspondante, contrairement à ce que fait free.

- Pour allouer un unique objet : **new type**
- Pour allouer un tableau de n objets : **new type[n]**

L'opérateur delete restitue la mémoire dynamique :

- Dans le cas d'un objet unique : **delete nom_var**
- Dans le cas d'un tableau de n objets : **delete [] nom_tab**

Travaux Pratique (TP 3)

```
- Donnez le nombre des matiÞres (0<N<=10) :  4
        1 - Entrez SVP [Nom Mat] [Coef] [Note C1] [Note C2]  :   Tech 1 12 10
        2 - Entrez SVP [Nom Mat] [Coef] [Note C1] [Note C2]  :   Math 2 16 12
        3 - Entrez SVP [Nom Mat] [Coef] [Note C1] [Note C2]  :   Algo 3 10 08
        4 - Entrez SVP [Nom Mat] [Coef] [Note C1] [Note C2]  :   Electronique 2 10 13
```

Travaux Pratique (TP 3)

-- IGA Marrekech --

Note du Semestre 1

	Coef	C1	C2	MOY
Tech	1	12.00	10.00	11.00
Math	2	16.00	12.00	14.00
Algo	3	10.00	8.00	9.00
Electronique	2	10.00	13.00	11.50
Moyenne Generale				11.13

LE LANGAGE C++

Les Classes

Notion orienté objet

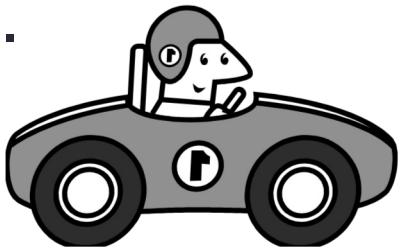
*De manière superficielle, le terme « **orienté objet** » signifie que l'on organise le logiciel comme une collection d'objets dissociés comprenant à la fois une **structure de données** - attributs - et un **comportement** – méthodes - dans une même entité.*

Notion orienté objet

Exemple :

une voiture peut avoir une certaine couleur et en même temps possède un comportement qui sera le même pour toutes les autres voitures, comme accélérer.

Ce concept est différent de la programmation conventionnelle dans laquelle les structures de données et le comportement ne sont que faiblement associés.



Les objets :

Chaque objet possède une identité et peut être distingué des autres. Deux pommes ayant les mêmes couleur, forme et texture sont des pommes individuelles. De la même façon, des jumeaux sont deux personnes distinctes, même si elles se ressemblent

les objets peuvent être distingués grâce à leurs **existences inhérentes** et **non grâce à la description des propriétés** qu'ils peuvent avoir.

Les objets :

Nous utiliserons l'expression « **instance** » d'objet pour faire référence à une chose précise, et l'expression « **classe** » d'objets pour désigner un groupe de choses similaires.

En d'autres termes, deux objets sont distincts même si tous leurs **attributs** (nom, taille et couleur par exemple) ont des valeurs identiques (deux pommes vertes sont deux objets distincts).

Les classes :

La classification signifie que les objets ayant la même structure de donnée – **attributs** - et le même comportement - **méthodes** - sont regroupés en une classe.

Les objets d'une classe ont donc le même type de comportement et les mêmes attributs.

Les méthodes peuvent être écrites une fois par classe, de telle façon que tous les objets de la classe bénéficient de la réutilisation du code.

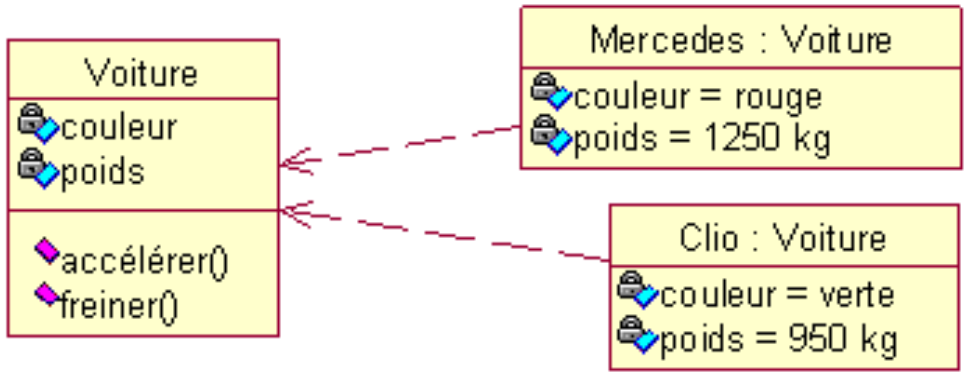
Les classes :

Par exemple, toutes les ellipses partagent les procédures de dessin, de calcul d'aire ou de test d'intersection avec une ligne.

Une classe est un modèle utilisé pour créer plusieurs objets présentant des caractéristiques communes.

Les classes :

Chaque objet possède ses propres valeurs pour chaque attribut mais partage noms d'attributs et méthodes avec les autres objets de la classe. Par exemple une classe Voiture peut être définie comme ayant un des attributs couleur et comme une des méthodes freiner ; les objets associées à la classe voiture peuvent être : Mercedes rouge, Clio verte, etc. Ces objets ont le même comportement (freiner).



Les attributs :

Un attribut est une valeur de donnée détenue par les objets de la classe. Couleur et Poids sont des attributs des objets relatifs à Voiture.

Chaque attribut à une valeur pour chaque instance d'objet. Par exemple, l'attribut Couleur porte la valeur rouge dans l'objet Mercedes alors que pour l'objet Clio, la valeur de l'attribut Couleur est verte. Les instances peuvent avoir des valeurs identiques ou différentes pour un attribut donné. Chaque nom d'attribut est unique à l'intérieur d'une classe.

Les méthodes :

Une méthode est une fonction ou une opération qui peut être appliquée aux objets ou par les objets dans une classe. Accélérer et freiner sont des méthodes de la classe Voiture. Tous les objets d'une même classe partagent les mêmes méthodes.

Encapsulation

L'encapsulation est le principe qui permet de regrouper les attributs et méthodes au sein d'une classe.

Cette notion est aussi associée au dispositif de protection qui permet de contrôler la visibilité d'un attribut ou d'une méthode. En d'autres termes, cela signifie que chaque fois que vous définissez un membre d'une classe (attribut ou méthode), vous devez indiquer les droits d'accès quant à l'utilisation de ce membre.

Encapsulation

Ce mécanisme d'encapsulation permet surtout de protéger l'objet de toute malveillance externe. Pour cela, la plupart du temps, il faut interdire l'accès direct aux attributs et passer systématiquement par les méthodes.

Par exemple : si l'on désire changer la couleur d'une voiture, cela ne se fait pas par enchantement, il est nécessaire de passer par une méthode de modification, et dans ce cas de figure l'attribut couleur ne doit pas être accessible directement.

Encapsulation

Comme nous venons de l'évoquer, le changement d'état d'un objet passe nécessairement par la méthode associée. Cela n'a aucun sens de vouloir atteindre directement un attribut.

Il existe trois niveaux de protection :

- **Public**
- **Private**
- **Protected**

Encapsulation : Public

Tous les attributs ou méthodes d'une classe définies avec le mot clé public sont utilisables par tous les objets. Il s'agit du niveau le plus bas de protection. Ce type de protection est employé pour indiquer que vous pouvez utiliser sans contrainte les attributs et les méthodes d'une classe.

Encapsulation : Private

Tous les membres d'une classe définis avec le mot clé private sont utilisables uniquement par les méthodes de la classe. Cette étiquette de protection constitue le niveau le plus fort de protection. Généralement les attributs doivent être déclarés comme privées.

Encapsulation : Protected

Tous les membres d'une classe définis avec le mot clé protected sont utilisables uniquement par les méthodes de la classe et par les méthodes des classes dérivées (par les enfants). Cette technique de protection est fortement associée à la notion d'héritage (voir ultérieurement).

Encapsulation

Par défaut, une structure possède le niveau de protection le plus faible, c'est-à-dire que tous les membres sont **publiques**. Il faut donc rajouter à notre implémentation les qualificatifs nécessaires pour améliorer la protection et donc respecter le contrat d'encapsulation.

En fait, pour être sûr de respecter le principe d'encapsulation, il est généralement préférable d'utiliser une structure qui est privée par défaut. Il s'agit de la structure **class**. Dorénavant, nous utiliserons le mot réservé **class** plutôt que **struct**.

Encapsulation

Tous les membres d'une classe doivent être au moins déclarés à l'intérieur de la formule **class*nom*{...}** qui constitue la déclaration de la classe.

Cependant, dans le cas des fonctions, aussi bien publiques que privées, on peut se limiter à n'écrire que leur en-tête à l'intérieur de la classe et définir le corps ailleurs, plus loin dans le même fichier ou bien dans un autre fichier.

Encapsulation

Il faut alors un moyen pour indiquer qu'une définition de fonction, écrite en dehors de toute classe, est en réalité la définition d'une fonction membre d'une classe. Ce moyen est **l'*opérateur de résolution de portée***, dont la syntaxe est :

***TypeRetour NomDeClasse*::NomFonction (……)**

Encapsulation

Par exemple, voici notre classe Point avec la fonction distance définie séparément :

```
class Point {
 int x;
 int y;
public:
...
double distance(Point autrePoint);
...
}
```

Il faut alors, plus loin dans le même fichier ou bien dans un autre fichier, donner la définition de la fonction promise dans la classe Point. Cela s'écrit :

```
double Point::distance(Point autrePoint)
{
    int dx = x - autrePoint.x;
    int dy = y - autrePoint.y;
   return sqrt(pow(dx,2)+ pow(dy,2));
}
```

Encapsulation

un programme classique. Dans l'exemple suivant,

```
// scope de la classe
class foo {
private :
            int i;
public :
            void g() {i = 3;};
}
```

```
// scope programme
void f()
{
            foo x;
            // x.i = 9; // Erreur
            x.g();      // Ok
}
```

Travaux Pratique (TP 1)

Écrire un programme qui permet d'introduire les informations suivantes pour une voiture :

- Une voiture est identifié par (Matricule et un Poids).
- Les attributs doivent être privés.

Crée une méthode qui permet de modifier le matricule d'une voiture, et une autre pour le poids.

La méthode afficher permet d'afficher les informations de la voiture (doit être externe a la classe)

Écrire un programme principale qui gère les traitements

Travaux Pratique (TP 2)

Dans la gestion d'une banque, chaque client doit avoir un compte bancaire, le compte bancaire est identifié par un *code* et un *solde*.

- Créer la classe Compte_B
- Créer les méthodes suivants :
  - Consulter (); *//affiche les informations du compte*
  - Retirer(double S); *//retirer une somme S*
  - Déposer(double S); *//ajoute une somme S au solde*

La méthode get()

Une méthode qui permet de retourner un attribut de la classe. Exemple :

```
class Voiture
{
        long Matr;
  public :
        long getMatr(){return Matr;}
}
```

La méthode set()

Une méthode qui permet la modification d'un attribut de la classe. Exemple :

```
class Voiture
{
        long Matr;
  public :
        void setMatr(long M){Matr = M;}
}
```

Les méthodes geters() et seters()

- Exemple :

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

class Voiture
{
  private :
          long Matr;
          int Poids;
  public :
          long getMatr(){return Matr;}
          int getPoids(){return Poids;}
          void setMatr(long M){Matr=M;}
          void setPoids(int P){Poids=P;}
          void afficher();
};
```

Le Pointeur this

On a parfois besoin de désigner à l'intérieur d'une fonction membre l'objet qui est manipulé par la méthode.

Comment le désigner cependant alors qu'il n'existe aucune variable le représentant dans la fonction membre? Les fonctions membres travaillent en effet directement sur les attributs de classes : ceux qui sont atteints correspondent alors à ceux de l'objet courant.

Le Pointeur this

C++ apporte une solution à ce problème en introduisant le mot-clé **this** qui permet à tout moment dans une fonction membre d'accéder à un pointeur sur l'objet manipulé.

Le Pointeur this

Exemple :

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

class Voiture
{
  private :
           long Matr;
           int Poids;
  public :
           void setMatr(long Matr){this->Matr=Matr;}
           void setPoids(int Poids){this->Poids=Poids;}
           void afficher();
};
```

L'opérateur « ***this*** » est justement un pointeur vers l'objet sur lequel nous travaillons.

Travaux Pratique (TP 3)

Partie 1 :

Créer une classe Point ayant les informations suivantes :

- X : Abscisse d'un point.
- Y : Ordonnée d'un point.
- Les méthodes geters et seters.
- Une méthode afficher qui permet de donner le résultat suivant :

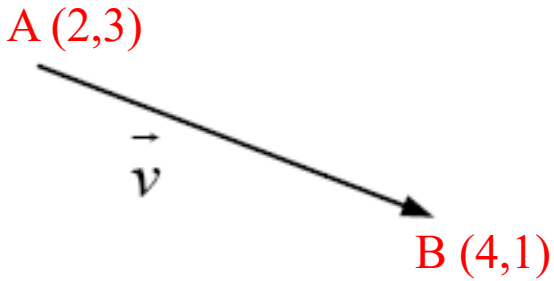
```
->   l'abscisse et l'ordonnee du point sont : [2,3].
```

Travaux Pratique (TP 3)

Partie 2 :

Créer une classe Vecteur ayant les informations suivantes :

- A : Premier point du vecteur.
- B : deuxième point du vecteur.
- Les méthodes geters et seters.
- Une méthode afficher tel que :




```
- les informations d'un vecteur sont :
        ->   l'abscisse et l'ordonnee du point sont : [2,3].
        ->   l'abscisse et l'ordonnee du point sont : [4,1].
```

```
->   la norme du vecteur est :  2.83
```

- Une méthode norme tel que :

$$\| \overrightarrow{AB} \| = \sqrt{X_{\overrightarrow{AB}}{}^2 + Y_{\overrightarrow{AB}}{}^2} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Constructeur

Vu le principe d'encapsulation, il est formellement interdit d'atteindre directement les attributs d'une classe. Il faut systématiquement passer par une méthode.

Du coup, il n'est pas possible de réaliser une initialisation explicite. Ne vous inquiétez pas, il existe des méthodes adaptées pour résoudre les initialisations explicites pour que l'objet soit dans l'état désiré. Ces méthodes spécifiques s'appellent des constructeurs.

Constructeur

Un constructeur est une méthode membre spéciale qui sera appelée automatiquement à chaque création d'un objet et qui :

- a le même nom que la classe,
- n'indique pas de type de retour,
- ne contient pas d'instruction return.

Constructeur

Le rôle d'un constructeur est d'initialiser un objet, notamment en donnant des valeurs à ses données membres.

Le constructeur n'a pas à s'occuper de trouver l'espace pour l'objet ; il est appelé (immédiatement) après que cet espace ait été obtenu, et cela quelle que soit la sorte d'allocation qui a été faite : statique ou dynamique, cela ne regarde pas le constructeur.

Constructeur

Exemple 1 :

```
class Point
{
   private:
            int x, y;
   public:
            Point() {x = 0; y = 0;}
            ... autres fonctions membres ...
};

int main()
{
            Point P;
            ... autres traitements ...
}
```

Constructeur

Un constructeur est une méthode, donc, lorsque nous créons un objet, nous devons faire appel **explicitement** à cette méthode, en utilisant la syntaxe habituelle de l'appel des fonctions, c'est-à-dire en utilisant les parenthèses. Pour le nombre complexe, nous avons besoin de préciser les deux arguments vu la signature du constructeur que nous avons choisi.

Constructeur

Exemple 2 :

```
class Point
{
   private:
            int x, y;
   public:
            Point(int a, int b) {x = a; y = b;}
            ... autres fonctions membres ...
};

int main()
{
            Point P(2,6);
            ... autres traitements ...
}
```

Constructeur

Une classe peut posséder plusieurs constructeurs, qui doivent alors avoir des signatures différentes :

```
class Point
{
   private:
            int x, y;
   public:
      Point(int a, int b) {x = a; y = b;}
      Point(int a) {x = a; y = 0;}
      Point() {x = y = 0;}
      ...
};
```

Constructeur

Comme les autres fonctions membres, les constructeurs peuvent être déclarés dans la classe et définis ailleurs.

```
class Point
{
   private:
            int x, y;
   public:
      Point(int , int );
      …
};
Point::Point(int a, int b)
{
      x = a; y = b;
}
```

Appel des constructeurs

Un constructeur est toujours appelé lorsqu'un objet est crée, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a(3, 4);
Point b = Point(5, 6);
```

dans le cas d'un constructeur avec un seul paramètre, on peut aussi adopter une forme qui rappelle l'initialisation des variables de types primitifs (à ce propos voir aussi la section 3.3.1) :

```
Point e = 7; // équivaut à : Point e = Point(7)
```

Appel des constructeurs

Un objet alloué dynamiquement est lui aussi toujours initialisé, au mois implicitement. Dans beaucoup de cas il peut, ou doit, être initialisé explicitement. Cela s'écrit :

```
Point *pt = new Point(1, 2);
```

Constructeur

Dans une classe il peut y avoir plusieurs constructeurs :

- **Un constructeur par défaut**
- **Un constructeur avec arguments**
- **Un constructeur de recopie**

Constructeur par défaut

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètres : ou bien il n'en a pas, ou bien tous ses paramètres ont des valeurs par défaut. Il joue un rôle remarquable, car *il est appelé chaque fois qu'un objet est crée sans qu'il y ait appel explicite d'un constructeur*, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité :

```
Point x;                    // équivaut à : Point x = Point()
Point t[10];                // produit 10 appels de Point()
Point *p = new Point;       // équivaut à : p = new Point()
Point *q = new Point[10];   // produit 10 appels de Point()
```

Constructeur avec arguments

Le constructeur avec arguments est un constructeur qui peut être appelé avec paramètres. Dans le cas d'un constructeur avec un seul paramètre, on peut aussi adopter une forme qui rappelle l'initialisation des variables.

```
Point a(3, 4);
Point b = Point(5, 6);
Point e = 7;        // équivaut à : Point e = Point(7)
```

Constructeur de recopie (ou par copie)

Le constructeur par copie d'une classe *C* est un constructeur dont le paramètre est de type *&C*. Ce constructeur est appelé lorsqu'un objet est initialisé en copiant un objet.

```
Point a;
Point b(5, 6);
Point e = Point(b);
Point f(e);
```

Exercice

- Soit la classe Date, contenant 3 entier (jj,mm,aa). Implémenter la classe par les méthodes suivantes :

  - les getters et les setters.
  - Les trois constructeurs, un par défaut le deuxième avec argument et le troisième avec recopie.
  - la méthode affichage qui permet d'afficher la date complète. Exemple : aa: 2011, mm=1, jj=22, l'affichage sera sous la forme suivante: 22 janvier 2011.

NB : pour les chaines de caractères, le type string est préférable a la déclaration char*.

Le destructeur

Le mécanisme automatique d'initialisation proposé par les constructeurs est très pratique. Mais d'autres problèmes surviennent : à quel moment libérer l'espace mémoire alloué pour stocker le nom d'un article en particulier? C++ apporte également une solution à ce problème grâce à un second mécanisme automatique, permettant d'appliquer une méthode particulière, appelée ***destructeur***, au moment de la destruction de l'objet.

Le destructeur

Quelques remarques sur cette méthode :

- il n'y a qu'un **seul** destructeur possible par classe ;
- l'identificateur désignant le destructeur est composé du nom de la classe préfixé par le symbole « ˜ » (ce qui donne ainsi le nom ˜Article pour désigner le destructeur de la classe Article) ;
- les destructeurs n'acceptent aucun paramètre et ne renvoient aucun résultat ;
- les destructeurs, comme les constructeurs, ne sont **jamais** explicitement appelés dans un programme. Ils sont utilisés automatiquement et implicitement par le compilateur au moment de la création et de la destruction des objets

Le destructeur

Exemple :

```
class Point
{
   private:
            int x, y;
   public:
      Point(int a, int b);
      {
      x = a; y = b;
      }
      ~Point();
      …
};
```

Travaux Pratique (TP 4)

- Créez la classe Technicien, possédera 6 attributs (**matricule**, **nom**, **prenom**, **salaire_par_heure**, **nombre_heures** , **date_embauche**) de visibilité privés.
  - Implémenter les getters et les setters.
  - Implémentez trois constructeurs, un par défaut le deuxième avec argument et le troisième avec recopie.
  - Implémentez la méthode affichage qui permet d'afficher les informations du salarié.
  - Implémentez la méthode CalculerSalaire().
  - Ajouter un destructeur contenant un messager, pour affirmer la destruction de l'objet.

Babysits

NB: utiliser la classe Date comme type pour la déclaration de la variable date_embauche

Amis

On a vu, par utilisation des sections public et private, que le C++ intègre totalement l'encapsulation des données. On peut ainsi interdire tout accès direct sur les informations maintenues par une classe.

Amis

Un mécanisme permet de casser totalement cette protection : les *amis.* Ce concept d'amitié est implémenté par utilisation du mot clé friend.

Ce qualificatif permet de déclarer qu'une entité étrangère à la classe (donc de provenance externe par rapport au scope de la classe) pourra avoir les autorisations de connaissance totale du scope de la classe courante.

Amis

Ce concept peut s'appliquer à une fonction globale programme, à une fonction membre particulière d'une autre classe, ou à une classe toute entière. Dans ce cas, si la fonction amie manipule une instance de cette classe, elle pourra modifier les données privées de cette classe.

Amis (Fonctions amies)

Il est parfois nécessaire d'avoir des fonctions qui ont un accès illimité aux champs d'une classe. En général, l'emploi de telles fonctions traduit un manque d'analyse dans la hiérarchie des classes.

Une fonction amie d'une classe est une fonction qui, sans être membre de cette classe, a le droit d'accéder à tous ses membres, aussi bien publics que privés.

Amis (Fonctions amies)

De telles fonctions sont appelées des *fonctions amies*. Pour qu'une fonction soit amie d'une classe, il faut qu'elle soit déclarée dans la classe avec le mot-clé friend.

Amis (Fonctions amies)

Les fonctions amies se déclarent en faisant précéder la déclaration de la fonction classique du mot-clé *friend* à l'intérieur de la déclaration de la classe. Les fonctions amies ne sont pas des méthodes de la classe cependant (cela n'aurait pas de sens puisque les méthodes ont déjà accès aux membres de la classe).

Amis (Fonctions amies)

```
class Tableau
{
    int *tab, nbr;
    friend void afficher(Tableau);
public:
    Tableau(int nbrElements);
};
```

```
void afficher(Tableau t)
{
    cout << '[';
    for (int i = 0; i < t.nbr; i++)
            cout << ' ' << t.tab[i];
    cout << ]" ;"
}
```

Notez cet effet de la qualification *friend* : bien que déclarée à l'intérieur de la classe Tableau, la fonction afficher n'est pas membre de cette classe ; en particulier, elle n'est pas attachée à un objet, et le pointeur **this** n'y est pas défini.

Amis (Fonctions amies)

En effet, dans la plupart des cas, une fonction amie peut avantageusement être remplacée par une fonction membre :

```
class Tableau
{
    int *tab, nbr;
public:
    Tableau(int nbrElements);
    void afficher();
};
```

```
void Tableau::afficher()
{
    cout << '[';
    for (int i = 0; i < nbr; i++)
            cout << ' ' << tab[i];
    cout << ]" ;"
}
```

Amis (Fonctions amies)

Il y a cependant des cas de figure où une fonction doit être nécessairement écrite comme une amie d'une classe et non comme un membre. un de ces cas est celui où la fonction doit, pour des raisons diverses, être membre d'une autre classe.

Imaginons, par exemple, que la fonction afficher nécessite l'écriture des éléments d'un objet Tableau dans un certain objet Fenêtre

Amis (Fonctions amies)

```
class Fenetre {
…
public:
void afficher(Tableau);
...
};
```

```
class Tableau {
int *tab, nbr;
public:
friend void Fenetre::afficher(Tableau);
...
};
```

Maintenant, la fonction afficher est *membre* de la classe Fenetre et *amie* de la classe Tableau : elle a tous les droits sur les membres privés de ces deux classes.

Amis

Donc il est possible de déclarer amie une fonction d'une autre classe, en précisant son nom complet à l'aide de l'opérateur de résolution de portée.

Mais il est aussi possible de rendre toutes les méthodes d'une classe amies d'une autres. C'est le concept de *classes amies*.

Amis (Classes amies)

Pour rendre toutes les méthodes d'une classe amies d'une autre classe, il suffit de déclarer la classe complète comme étant amie. Pour cela, il faut encore une fois utiliser le mot-clé *friend* avant la déclaration de la classe, à l'intérieur de la classe cible.

Amis (Classes amies)

Une classe amie d'une classe *C* est une classe qui a le droit d'accéder à tous les membres de *C*. Une telle classe doit être déclarée dans la classe *C* (la classe qui accorde le droit d'accès), précédée du mot réservé *friend*, indifféremment parmi les membres publics de *C*.

Amis (Classes amies)

```
class Point
{
   private:
          int x, y;
   public:
     Point() {x = y = 0;}
     friend class Triangle;
     ...
};
```

```
class Triangle
{
   private:
          Point a,b,c;
   public:
     Triangle(Point a, Point b, Point c)
     {
          this→a.x=a.x;
          this→a.y=a.y;
          …
     }
     ...
};
```

Amis

Attention :

- la notion d'amitié est contraire a la notion d'encapsulation. Elle doit être utiliser de manière pertinente et avec précaution.
- La relation d'amitié n'est pas transitive, « les amis de mes amis ne sont pas mes amis ».

Travaux Pratique (TP 5)

Créer une classe Point ayant les informations suivantes :

- X : Abscisse d'un point.
- Y : Ordonnée d'un point.
- Un constructeur par défaut et un autre avec arguments.
- Une fonction *MoveTo* et une classe *Vecteur*, amies de cette classe.

Créer une classe Vecteur ayant les informations suivantes :

- Deux points A et B.
- Un constructeur avec arguments.
- Une fonctions afficher amie de cette classe.
- Que remarquez-vous ?. Proposez la solution la plus simple.

```
x : [ 2 , 8 ] , y : [ 4 , 7 ]
```

LE LANGAGE C++

Surcharge des opérateurs

Surcharge des opérateurs

En C++ on peut redéfinir la sémantique des opérateurs du langage, soit pour les étendre à des objets, alors qui n'étaient initialement définis que sur des types primitifs, soit pour changer l'effet d'opérateurs prédéfinis sur des objets. Cela s'appelle ***surcharger des opérateurs***.

Il n'est pas possible d'inventer de nouveaux opérateurs ; seuls des opérateurs déjà connus du compilateur peuvent être surchargés.

Surcharge des opérateurs

Une fois surchargés, les opérateurs gardent leur pluralité, leur priorité et leur associativité initiales.

Surcharger un opérateur revient à définir une fonction, tout ce qui a été dit à propos de la surcharge des fonctions s'applique donc à la surcharge des opérateurs.

Surcharge des opérateurs

Plus précisément, pour surcharger un opérateur X (le signe X représente un opérateur quelconque) il faut définir une fonction nommée ***operatorX***. Ce peut être une fonction membre d'une classe ou bien une fonction indépendante.

Si elle n'est pas membre d'une classe, alors elle doit avoir au moins un paramètre d'un type classe.

Surcharge des opérateurs (fonction membre)

Si la fonction operator*X* est membre d'une classe, elle doit comporter un paramètre de moins que la pluralité de l'opérateur : le premier opérande sera l'objet à travers lequel la fonction a été appelée. Ainsi, sauf quelques exceptions :

- « *objX* ou *Xobj* équivalent à *obj*.operator*X*() »
- « *obj*1 *X* *obj*2 équivaut à *obj*1.operator*X*(*obj*2) »

Surcharge des opérateurs (fonction membre)

Exemple :

```
class Point {
      int x, y;
public:
      Point(int x= 0, int y= 0){
        this→x=x; this→y=y;}
      int Getx() { return x; }
      int Gety() { return y; }
      Point operator+(Point);
      …
};
Point Point::operator+(Point q){
return Point(x + q.x, y + q.y);
}
```

```
Point p, q, r;
...
r = p + q;      // compris comme : r = p.operator+(q);
```

Surcharge des opérateurs (fonction membre)

```
Point p, q, r;
int x, y;
```

Surcharge de l'opérateur + par une fonction membre :

```
r = p + q;   // Oui : r = p.operator+(q);
r = p + y;   // Oui : r = p.operator+(Point(y));
r = x + q;   // Erreur : x n'est pas un objet
```

Surcharge des opérateurs
(fonction membre)

Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable. Une fonction membre renforce l'encapsulation. Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage

Surcharge des opérateurs
(fonction non membre)

Si la fonction operatorX n'est pas membre d'une classe, alors elle doit avoir un nombre de paramètres égal à la pluralité de l'opérateur. Dans ce cas :

- *« objX ou Xobj équivalent à operatorX(obj) »*
- *« obj1 X obj2 équivaut à operatorX(obj1, obj2) »*

Surcharge des opérateurs (fonction non membre)

Exemple :

```
class Point {
      int x, y;
public:
      Point(int x= 0, int y= 0){
        this→x=x; this→y=y;}
      …
};
Point operator+(Point p , Point q){
return Point(p.Getx() + q.Getx(), p.Gety() + q.Gety());
}
```

```
Point p, q, r;
...
r = p + q;      // compris comme : r = operator+(p,q);
```

Surcharge des opérateurs
(fonction non membre)

```
Point p, q, r;
int x, y;
```

Surcharge de l'opérateur + par une fonction non membre :

```
r = p + q;   // Oui : r = operator+(p,q);
r = p + y;   // Oui : r = operator+(p,Point(y));
r = x + q;   // Oui : r = operator+(Point(x),q);
```

Surcharge des opérateurs
(fonction non membre)

Lorsque la surcharge d'un opérateur est une fonction non membre, on a souvent intérêt, ou nécessité, à en faire une fonction amie. Par exemple, si la classe Point n'avait pas possédé les accesseurs publics Getx() et Gety(), on aurait dû surcharger l'addition par une fonction amie

Surcharge des opérateurs
(fonction non membre)

Exemple :

```
class Point {
int x, y;
public:
Point(int, int);
friend Point operator+(Point, Point);
...
};
Point operator+(Point p, Point q) {
return Point(p.x + q.x, p.y + q.y);
}
```

Surcharge des opérateurs
(fonction non membre)

La surcharge d'un opérateur binaire par une fonction non membre est carrément obligatoire lorsque le premier opérande est d'un type standard ou d'un type classe défini par ailleurs, que le programmeur ne peut plus étendre

Travaux Pratique (TP 6)

Reprenez la classe Point avec un constructeur avec arguments et surchargez les opérateurs suivants :

- + , - , * avec des fonction non membre de la classe.
- = ,+= , -= , = = , < , >= , ++ et -- (préfixé) , ++ et -- (postfixé)

Avec une fonction *afficher()* pour voir le résultat.

Injection et extraction de données

- L'opérateur **<<** peut être utilisé pour « injecter » des données dans un flux de sortie, ou *ostream.*
- L'opérateur **>>** peut être utilisé pour « extraire » des données du un flux d'entré, ou *istream.*

Les deux opérateurs peuvent être surchargé pour leurs utilisation avec des objets.

La surcharge des deux opérateurs se fait que par une fonction non membre (de préférence amie)

Injection et extraction de données

```
class Point {
...
friend ostream& operator<<(ostream&, Point);
};
ostream& operator<<(ostream &o, Point p) {
return o << '(' << p.x << ',' << p.y << ') '<<endl;
}
```

```
Point p;
...
cout << "le point trouvé est : " << p ;
```

Injection et extraction de données

On peut de manière analogue surcharger l'opérateur >> afin d'obtenir une moyen simple pour lire des points. Par exemple, si on impose que les points soient donnés sous la forme (*x; y*), c'est-à-dire par deux nombres séparés par une virgule et encadrés par des parenthèses :

```
class Point {
...
        friend istream& operator>>(istream&, Point&);
};
istream& operator>>(istream& i, Point& p) {
    char c;
    i >> c;
    if (c == '(') {
        cin >> p.x >> c;
        if (c == ',') {
            cin >> p.y >> c;
            if (c == ')')
                return i;
        }
    }
    cerr << "Erreur de lecture. Programme avorté\n";
    exit(-1);
}
```

Injection et extraction de données

Exemple d'utilisation :

```
void main() {
Point p;
cout << "donne un point : ";
cin >> p;
cout << "le point donné est : " << p << "\n";
}
```

Essai de ce programme :

donne un point : *(2 , 3)*

le point donné est : (2,3)

Exercice

On souhaite développer une classe Temps qui contiendra :

- 3 variables (H, M, S).
- 2 constructeur, un avec arguments dont les valeurs sont par défaut, et l'autre par recopie.
- Surcharge de l'opérateur ++ (préfixé) tel que :
  - (22:58:58) → (22:58:59)
  - (22:58:59) → (22:59:00)
  - (22:59:59) → (23:00:00)
  - (23:59:59) → (00:00:00)
- Surcharge de l'opérateur d'extraction >> tel que : (EX H:M:S : 22:58:58)
- Surcharge de l'opérateur d'injection << qui affiche l'heure en format 12H tel que : (EX : 20:16:55) → 08:16:55 PM

LE LANGAGE C++

Les tableaux d'objets

Création de la classe

- L'utilisation d'un tableau d'objets d'une classe nécessite la création d'une classe spécial contenant deux attributs :
  - Pointeur : sur type d'objet de chaque composante du tableau
  - Un entier : représentant la dimension du tableau.

- Pour la classe Point La syntaxe est la suivante :

```
class Tpoint{
        point * tab;
        int nbr;
        ….
};
```

Constructeurs

- Avant de créer les constructeurs pour la classe Tpoint, il faut s'assurer que les constructeurs pour la classe point sont déjà créer et surtout le constructeur par défaut (Il n'y a pas moyen de spécifier des paramètres au constructeur dans la déclaration d'un tableau).

```
class Tpoint{
        ….
        Tpoint() {nbr=1;tab=new point[nbr];}
        Tpoint(int n) {nbr=n;tab=new point[nbr];}
        ….
};
```

Destructeur du tableau

- Le destructeur est obligatoire pour les classes tableaux, ce destructeur permet de détruire le pointeur au niveau de la déclaration.

- La syntaxe est la suivante :

```
class Tpoint{
        ….
        ….
        ~Tpoint(){delete [] tab;}
        ….
};
```

Surcharge de l'opérateur [ ]

Comme les opérateurs déjà vu dans le chapitre précédant, l'opérateur [ ] (utilisé pour les tableaux) peut aussi être surchargé.

La syntaxe est la suivante :

```
class Tpoint{
        ….
        ….
        point& operator[](int ind)
        {return tab[ind];}
        ….
};
```

Déclaration au niveau du main

Au niveau de la fonction principale main, la déclaration d'un tableau de point prend un autre sens d'écriture.

Ce n'est pas comme le langage C, où il suffit de déclarer la variable et la dimension du tableau (EX : int T[20] ;). Pour le langage C++ le tableau lui-même est une classe, donc il vous faut d'appeler le constructeur de cette classe.

La syntaxe est la suivante :

```
main()
{
        Tpoint T1,T2(3);
}
```

La variable T1 est un tableau de points contenant une case (appel du constructeur par défaut de la classe Tpoint)

La variable T2 est un tableau de points contenant trois cases (appel du constructeur par arguments de la classe Tpoint)

Les algorithmes de création et d'affichage

- Les algorithmes de création et d'affichage ce base en général sur la surcharge d'opérateurs d'extraction et d'injection déjà surcharger pour la classe point.

```
class Tpoint{
        ….
        void Creation()
        {
              for(int i=0;i<nbr;i++)  cin>>tab[i];
        }
        void Afficher()
        {
              for(int i=0;i<nbr;i++)  cout<<tab[i];
        }
        ….
};
```

Les algorithmes d'ajout

- L'algorithme d'ajout est pratiquement pareille que l'algorithme de création, a une exception prêt c'est d'ajouter un seul élément.

```
class Tpoint{
        ….
        void Ajout(point p)
       {
            point * newtab=new point[nbr+1];
            for(int i=0;i<nbr;i++) newtab[i]=tab[i];
            nbr++;
            newtab[nbr-1]=p;
            tab= newtab;
       }
        ….
};
```

Les algorithmes de Recherche

- L'algorithme de la recherche nécessite la surcharge de l'opérateur == pour la classe point, et cela pour but de simplicité et d'efficacité.

```
class Tpoint{
        ….
        int Recherche(point p)
        {
            int pos=-1;
            for(int i=0;i<nbr && pos==-1;i++)
                if(tab[i]==p)
                  pos= i;
            return pos;
        }
        ….
};
```

Les algorithmes de modification

- L'algorithme de modification fait appel a la recherche et a la surcharge de l'opérateur d'extraction.

```
class Tpoint{
        ….
        void Modification(point p)
        {
                int pos=  Recherche(p);
                if(pos==-1)
                 cout<<"element n'existe pas "<<endl;
                else
                  cin>>tab[pos];
        }
        ….
};
```

Les algorithmes de suppression

- L'algorithme de suppression fait appel a la recherche.

```
class Tpoint{
        ….
        void Supprimer(point p)
           {
                int pos=  Recherche(p);
                if(pos==-1)
                        cout<<"element n'existe pas "<<endl;
                 else
                 {
                     for(int i=pos;i<nbr-1;i++)
                          tab[i]=tab[i+1] ;

                      nbr--;
                }
           }
        ….
};
```

Les algorithmes de trie

- L'algorithme de trie fait appel a la surcharge d'opérateur > ou bien <, = et de [].

```
class Tpoint{
        ....
         void Trie()
        {
                point aux;
                for(int i=0;i<nbr-1;i++)
                   for(int j=i+1;j<nbr;j++)
                     if(tab[i]>tab[j])
                     {
                             aux=tab[i];
                              tab[i]=tab[j];
                              tab[j]=aux;
                     }
        }
        ....
};
```

Exercice d'application

- Créer la classe Etudiant contenant les attributs suivant :
  - num, nom, prenom, note **(pour les chaine de caractère vous pouvez utiliser le type string)**
  - Ajouter les getters et les setters, puis le constructeur avec arguments donc les valeurs par défaut et un autre par recopie.
  - Surcharger les opérateurs = , == , > , >> , <<.
- Créer la classe TEtudiant (le tableau d'étudiant) :
  - Ajouter les méthodes création et affichage.
  - la méthode recherche qui permet de rechercher suivant le numéro d'étudiant.
  - la méthode modification qui permet de modifier l'étudiant.
  - La méthode suppression pour supprimer un étudiant.
  - la méthode trie pour trier les étudiant par note **(le type de trie est par choix).**
- Implémenter l'appel des méthodes dans la fonction main().

LE LANGAGE C++

L'Héritage

Définition de l'héritage

L'*héritage* permet de donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes. Les classes dont elle hérite sont appelées *classes mères*, *classes de base* ou *classes antécédentes*. La classe elle-même est appelée *classe fille*, *classe dérivée* ou *classe descendante*.

Les *propriétés héritées* sont les champs et les méthodes des classes de base.

Définition de l'héritage

Pour faire un héritage en C++, il faut faire suivre le nom de la classe fille par la liste des classes mères dans la déclaration avec les restrictions d'accès aux données, chaque élément étant séparé des autres par une virgule.

Définition de l'héritage

La syntaxe est la suivante :

```
class Classe_mere
{
/* Contenu de la classe mère. */
} ;
```

```
class Classe_fille : public|protected|private Classe_mere
{
/* Définition de la classe fille. */
} ;
```

Membres protégés

En plus des membres publics et privés, une classe *C* peut avoir des membres *protégés*. Annoncés par le mot clé **protected**, ils représentent une accessibilité intermédiaire car ils sont accessibles par les fonctions membres et amies de *C* et aussi par *les fonctions membres et amies des classes directement dérivées de C*.

Les membres protégés sont donc des membres qui ne font pas partie de l'interface de la classe, mais dont on a jugé que le droit d'accès serait nécessaire ou utile aux concepteurs des classes dérivées.

La signification des mots-clés private, protected et public dans l'héritage est récapitulée dans les schémas suivants :

Héritage privé

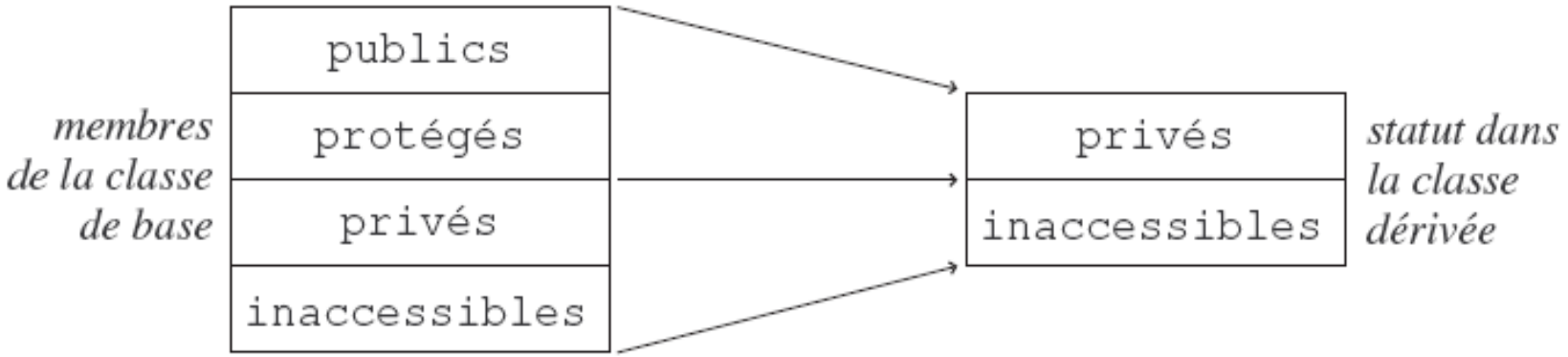


FIG. 5 – Héritage privé

Héritage protégé

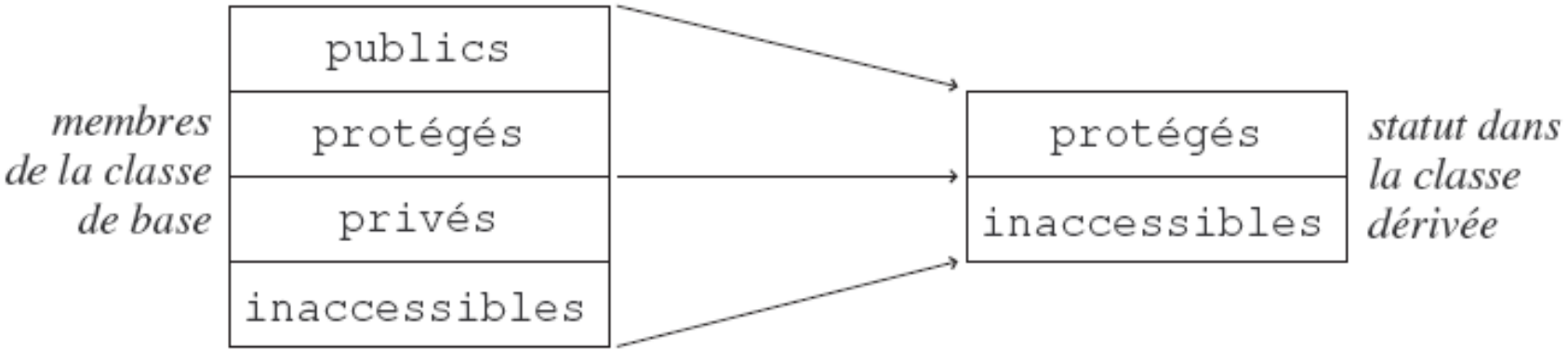


FIG. 6 – Héritage protégé

Héritage public

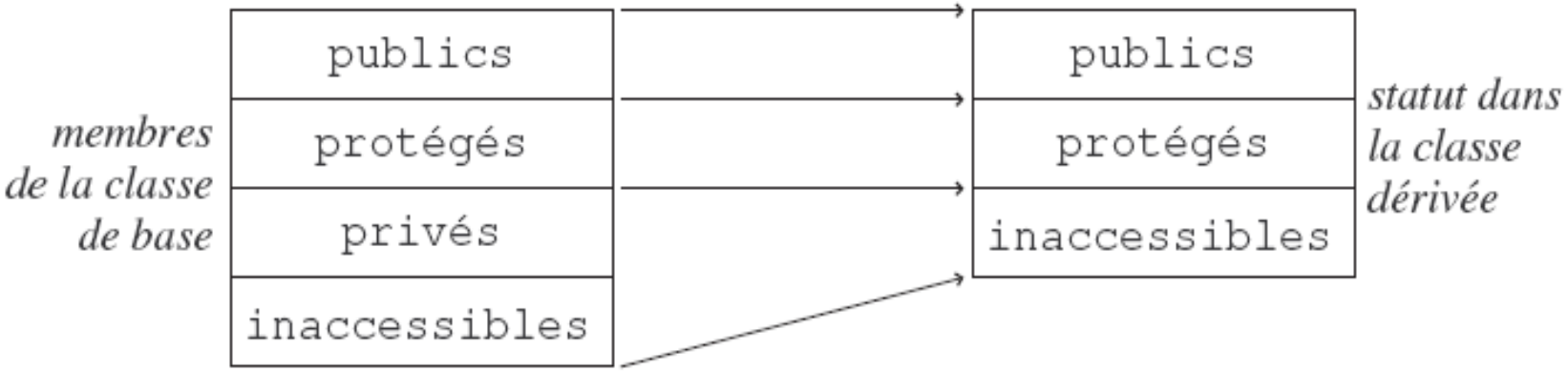


FIG. 7 – Héritage public

Définition de l'héritage

Le tableau suivant montre comment sont modifiés les contrôles d'accès en fonction du mode d'héritage.

héritage \ membres	`public`	`protected`	`private`
`public`	`public`	`protected`	Inaccessible
`protected`	`protected`	`protected`	Inaccessible
`private`	`private`	`private`	Inaccessible

Définition de l'héritage

Remarque :

- *Le mode d'héritage public est le plus utilisé. Il respecte le principe de substitution.*
- *Le mode d'héritage private est le mode par défaut en C++.*

Polymorphisme

Le *polymorphisme* est l'aptitude qu'on des objets à réagir différemment à un même message, une méthode commune à une hiérarchie de classe peut prendre plusieurs formes. Le choix de la méthode à appeler est retardé jusqu'à l'exécution du programme.

Polymorphisme

En polymorphisme il y'a deux notions fondamentales :

- **Compatibilité par affectation entre type de classe et un type ascendant.**
- **La redéfinition de méthodes.**

Polymorphisme

```
class Point {
protected :
int x, y;
public:
            Point(int,int):
            void afficher();
};
```

```
class Pixel : public Point {
string couleur;
public:
            Pixel(int, int, string);
            void afficher();
};
```

→ **Utilisation :**

```
Pixel px(1, 2, "rouge");
Point pt = px;        // Comatibilité par affectation entre type de classe et un type ascendant.

pt.afficher();        // La redéfinition de méthodes
```

Exercice

Créer un projet en C++, comportant les classes suivantes :

- Une classe **Point2D**, ayant comme attributs X et Y ainsi que la nomenclature de base (constructeur, geters, seters et une méthode d'affichage).
- Une classe **Point3D**, héritant de **Point2D** et ayant un autre attribut de plus Z, avec la redéfinition de la méthode d'affichage.
- Une classe **Point3DCouleur**, héritant de **Point3D** et ayant un autre attribut de plus de type string (intitulé de couleur), avec la redéfinition de la méthode d'affichage.

- Utilisez le même principe de l'héritage pour faire un programme qui permet de manipuler des points 2D, 3D et 3D en couleur. Tester aussi le principe de polymorphisme entre les classes

Méthode Virtuelle

Une *méthode virtuelle* est une méthode qui est déclarée définie ou non dans une classe, mais qui doit être redéfinie dans les classes dérivées de cette classe.

Il suffit de rajouter le mot-clé virtual devant la *déclaration* de la méthode. La méthode peut alors être redéfinie dans chaque sous-classe.

Méthode Virtuelle

Exemple :

```
class Point
{
    int x, y; // Attributs
            public:
                    virtual void affiche(){ cout << "x = " << x << ", y = " << y; }
// ...
};
```

Méthode Virtuelle Pure

Une méthode virtuelle pure est une méthode qui est déclarée mais non définie dans une classe. Elle est définie dans toutes les classes dérivées de cette classe.

Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « =0 ». La fonction doit également être déclarée virtuelle

(=0 signifie ici simplement qu'il n'y a pas d'instance de cette méthode dans cette classe).

Méthode Virtuelle Pure

Exemple :

```
class Point
{
    int x, y; // Attributs
            public:
                    virtual void affiche() =0;
// ...
};
```

Classes abstraites

Une classe est considérée comme abstraite si son interface contient au moins une *méthode virtuelle pure*.

Les classe abstraites sont *non instanciables* et servent uniquement de cadre de référence.

- Le compilateur C++ interdit d'instancier la classe et vérifie que tous les descendants redéfinissent la méthode.
- Les méthodes virtuelles pures peuvent être déclarées protected puisqu'elles ne sont destinées qu'à être redéfinies par les descendants.

Classes abstraites

Remarque :

- *Une classe ne possédant que des méthodes virtuelles pures est une spécification d'une **interface** (noms et signatures des opérations imposés pour toutes les classes d'implémentation).*
- *Il ne faut pas confondre **surcharger** (fonctions de même nom mais avec des listes de paramètres différentes) et **redéfinir** (fonction identique à une fonction héritée).*

Héritage et amitié

L’amitié pour une classe s’hérite, mais uniquement sur les membres de la classe hérités, elle ne se propage pas aux nouveaux membres de la classe dérivée et ne s’étend pas aux générations suivantes.

Héritage et amitié

Exemple :

```
Class Test1
{
        int x,y,z;
Public :
        friend void f(Test1);
        ……
};
```

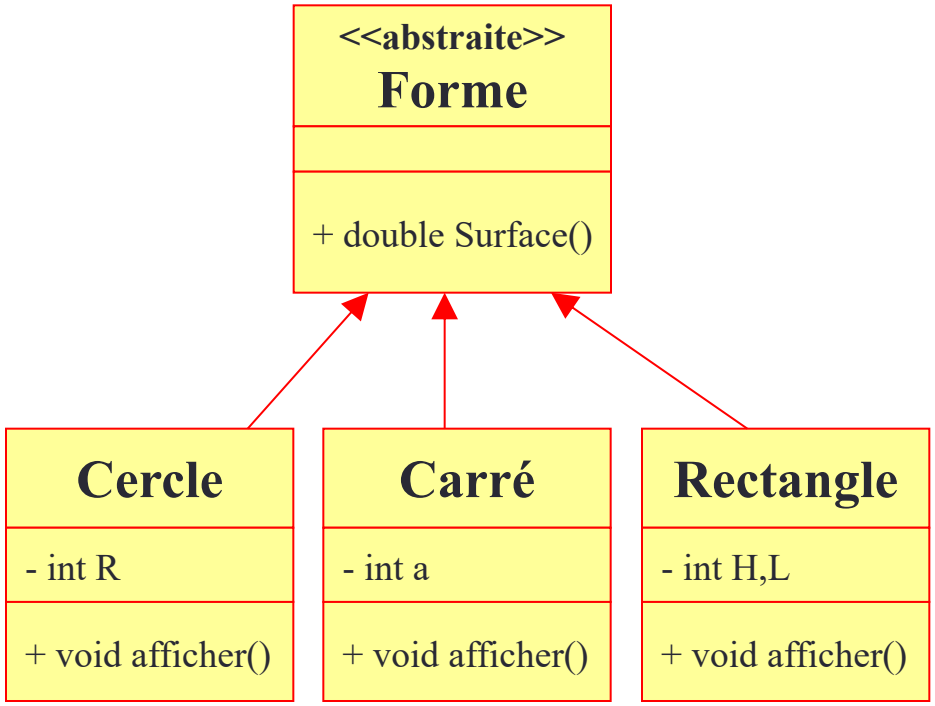
```
Class Test2 : public Test1
{
        ……
};
```

```
Test2 a;
f(a);       // OK, L'amitié pour une classe s'hérite
```

Travaux Pratique (TP 7)

Proposez un programme qui permet de calculer la surface d'un cercle, d'un carré et d'un rectangle.

Proposez une solution pour le calcule de la surface d'un cube.



Travaux Pratique (TP 8)

Une entreprise de textile souhaite informatiser le processus de paiement de ses employés. Pour cela, chaque employé est identifié par un code, un nom, un prénom et un numéro de téléphone. L'entreprise en question comporte :

- Un directeur, qui en plus des informations normales d'un employé, est identifié par le salaire mensuel ainsi qu'une observation sur sa responsabilité.
- Les autres employés doivent avoir l'intitulé du secteur sur lequel ils travaillent, il existe trois secteurs « Jeans », « Laines » et « la soie ». On peut compter deux types d'employés :
  - Un chef de secteur qui s'occupe de la bonne qualité des tissus ainsi que des pointages des couturières.
  - Une couturière dont le salaire dépend du nombre de pièces réalisés.

Règles de gestion :

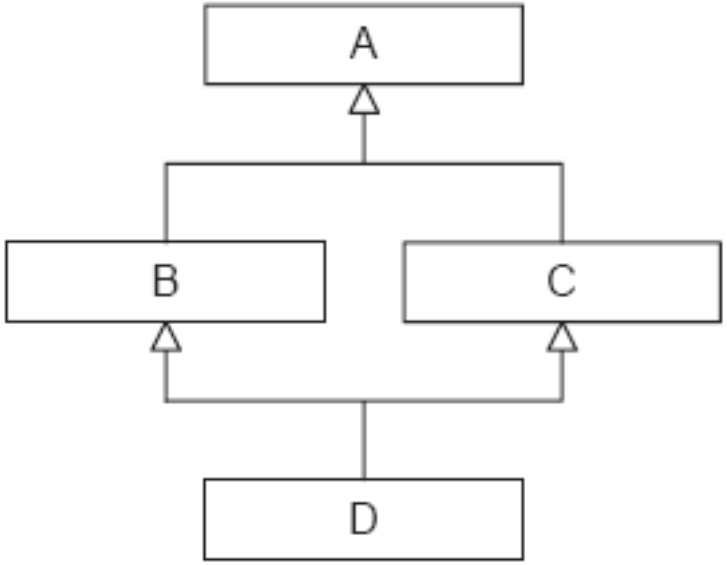
- Le chef de secteur est payé par heur, et peut effectuer des heurs supplémentaires.
- Un secteur doit comporter un chef et deux couturières.
- il faut retire 17% du salaire brut de chaque employé (valeur d'impôts IR « impôt sur le revenu »). Cette règle vous permet d'avoir le salaire net de chaque employé.

Héritage Multiple

Avec l'héritage simple,une classe peut hériter d'une classe et éventuellement par transitivité, d'autres classes. Mais une classes ne peut pas hériter de deux classes (ou plus). Ceci peut se faire, par contre, grâce à l'héritage multiple.

Héritage Multiple

Avec l'héritage multiple, une classe peut hériter plusieurs fois de l'un de ses ancêtres (héritage *à répétition*).



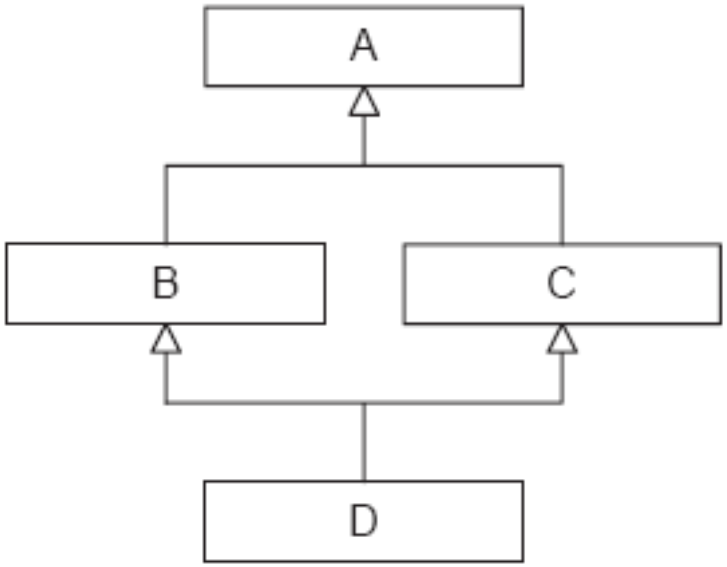
Héritage Multiple

Dans ce cas, les attributs de la classe A sont dupliqués dans la classe D. Il est possible de résoudre les ambiguïtés en préfixant le nom de l'attribut par le nom des classes. Par exemple, pour accéder à l'attribut **attr** déclaré dans la classe A mais correspondant à B, on écrit **B::attr**.

Héritage Multiple

Si l'on souhaite éviter la duplication des attributs, on déclare l'héritage virtuel.

```
class A { /* . . . */};
class B : virtual public A { /* . . . */};
class C : virtual public A { /* . . . */};
class D : public B, public C { /* . . . */};
```



Travaux Pratique (TP 9)

Une entreprise souhaite automatiser la gestion de paiement de ses employés. Chaque employé est identifié par le nom, le prénom, l'âge, la date d'embauche et le numéro de téléphone. L'entreprise en question regroupe quatre employés :

- Un technicien en informatique dont le salaire dépend du nombre d'unités réparées;
- Deux commerciaux qui ont une commission en plus du salaire mensuel;
- Un technicien commercial dont le salaire dépend des deux critères.

Pour chaque employé vous devez afficher le nom et le prénom, l'âge, la date, le téléphone et le salaire mensuel. C'est pourquoi vous devez disposer d'une fonction afficher (pour afficher les informations des employés), ainsi que d'une fonction virtuelle pure **calculeSalaire()** pour calculer le salaire pour chaque type d'employé,

En se basant sur l'analyse du sujet, vous êtes amené à réaliser un programme capable de résoudre les différentes contraintes posées par l'analyse du système d'information de l'entreprise.