Les Classes I

Licence de Physique

Table des matières

L'intérêt du langage C++ réside dans l'utilisation de ***Classes*** construites par vous même ou par d'autres utilisateurs et qui peuvent être très utiles. Cette notion de classe est en faite la généralisation de la *structure du C*.

1 Les Structures

Une structure en C permet de définir un type (au sens du C) plus complexe que les types de bases. C'est un ensemble de variables structurées regroupées sous une même "étiquette". Considérons par exemple un point dans le plan ; celui-ci a 2 composantes X et Y. Il est alors pratique de pouvoir définir un type unique `Point` comportant deux **champs** ou **attributs** X et Y :

```
#include <iostream>
using namespace std;
struct Point
{
  double X; // 1er champs
  double Y; // 2eme champs
};
```

On utilise alors ce nouveau type comme suit

```
int main()
{
  struct Point P1,P2; // on definit 2 variables P1 et P2 de  types Point
  P1.X=4.5; //on affecte le champs X de P1
  P1.Y=2.5; //on affecte le champs Y de P1
  P2.X=3.1; //on affecte le champ X de P2
  P2.Y=5.1; //on affecte le champ Y de P2
  cout<<"le point P1 a pour coordonnées : ("<<P1.X<<","<<P1.Y<<")"<<endl;
```

```
  cout<<"le point P2 a pour coordonnées : ("<<P2.X<<","<<P2.Y<<")"<<endl;
}
```

Remarque :

— On accède à un champs d'une structure par un point "." (par exemple P1.X).
— Les champs d'une structure peuvent avoir des types différents.
— Pour des variables différentes (ici P1 et P2), les noms des champs (ici X et Y) restent identiques.

2 Les classes

2.1 Introduction

On peut bien sûr utiliser une **classe** "*comme une structure*" (c'est ce que nous allons commencer par faire) mais cela va beaucoup plus loin ; plutôt que de définir des *types* (au sens du C) elles permettent de définir des ***objets***. Un objet c'est quelque choses qui possède des **champs de variables** (comme une structure) et des **méthodes** pour agir qui permettent de le manipuler (en particulier des méthodes pour le créer (ou *construire*) et le *détruire*).

La notion de **classe** est fondamentale en C++. On dit que le C++ est **un langage orienté objets**. Par rapport aux méthodes traditionnelles, la *"programmation objet"* est une nouvelle façon de programmer, car on va créer une classe (ou utiliser une classe existante) pour chaque "concept" qui semble ressortir du problème que l'on traite. Par exemple si l'on résout un problème d'algèbre linéaire, ce sera bien de programmer au préalable une classe de vecteurs et de matrices avec toutes leurs opérations standard. Le programme principal sera ainsi beaucoup plus simple en apparence, puisque on pourra écrire les opérations sous la forme W=M*N*V par exemple où V,W sont des vecteurs et M,N des matrices.

Un avantage de la programmation objet par rapport à la programmation habituelle, est donc que le code est **mieux structuré** et **plus lisible**. Un autre avantage est que le code est plus facilement **réutilisable** (Vous pouvez vous confectionner une bibliothèque de classes usuelles et/ou trouver des classes sur le web prêtes à l'emploi).

Remarque

— Au début, si vous n'êtes pas familier avec la notion d'objets, il vous sera plus facile de comprendre les classes comme un nouveau type que vous définissez ; **en particulier vous ne devez pas confondre une classe** ("le type", au même titre que **int** ou **double**) avec un **objet** de cette classe ("une variable"). Par exemple si vous avez défini une classe *MaClasse* et que vous déclarer *MaClasse obj* ; *obj* est la "variable" du "type" *MaClasse* (exactement comme dans *double x*, *x* est une variable de type *double*).

2.2 Ecriture d'une classe

Dans un premier temps, nous allons réécrire la structure Point sous la forme d'une classe.

```
#include <iostream>
using namespace std;
class  Point //on declare la classe Point
{
 public:
    double X; // 1er attribut de Point
    double Y; // 2eme attribut
}; //fin de la declaration de la classe. Noter le ;
int main()
{
  Point P1,P2; // on definit 2 OBJETS Point P1 et P2
  P1.X=4.5; //on affecte l'attribut X de P1
  P1.Y=2.5; //on affecte l'attribut Y de P1
  P2.X=3.1; //on affecte l'attribut X de P2
```

```
  P2.Y=5.1; //on affecte l'attribut Y de P2
  cout<<"le point P1 a pour coordonnées : ("<<P1.X<<","<<P1.Y<<")"<<endl;
  cout<<"le point P2 a pour coordonnées : ("<<P2.X<<","<<P2.Y<<")"<<endl;
  return 0;
}
```

Remarque :

— Le mot **class** suivit du nom indique la déclaration de la classe. Elle commence par une `{` et ce termine par `};` **Ne pas oublier le ;**

Dans une classe, on a des *membres*, qui sont soit des *attributs* (une variable) soit des *méthodes* ou *fonctions membres* (voir ci-après). Ces membres peuvent être **private** ou **public** [1]. Un membre **public** est directement accessible depuis l'extérieur de la classe (c'est le cas des attributs X et Y qui sont modifiés ou utilisé dans la fonction *main()*. Un membre **private** (privé) ne peut être référencé que par une méthode de la classe (voir ci-après).

— Dans la fonction *main()*, on construit un objet *P1* de la classe **Point** (exactement comme X est un objet de la classe **double**). Cela signifie qu'on réserve la place nécessaire pour cet objet. La syntaxe pour accéder à l'attribut *att* d'un objet *obj* est donc *obj.att*.

—

2.2.1 Constructeur, Destructeur et Méthodes

Reprenons l'exemple précédant en le modifiant un peu.

```
#include <iostream>
using namespace std;
class  Point //on declare la classe Point
{
 public:
    double X;
    double Y;
    Point(double n, double m)
    {
       X=n;
       Y=m;
       cout<<"Constructeur de Point"<<endl;
    }
    ~Point()
    {
       cout<<"Destructeur de Point"<<endl;
    }
    void Print()
    {
      cout<<"Point P("<<X<<","<<Y<<")"<<endl;
    }
}; //fin de la declaration de la classe. Noter le ;
int main()
{
       Point P1(4.5,2.5);
       Point P2;
       P2.X=3.1;
       P2.Y=5.1;
       P1.Print();
```

1. Il existe aussi une autre catégorie qui est **protected**, mais nous n'en parlerons pas.

```
        P2.Print();
        return 0;
}
```

Remarque :

— Nous avons rajouté 3 membres dans la classe : un **constructeur**, un **destructeur**, une **méthode** *Print()*.

1. le constructeur : il n'a **pas de type**, porte le **même nom que la classe** et peut avoir des arguments (comme ici, n et m).
2. le destructeur : il n'a **pas de type**, porte le **même nom que la classe précédé d'un tilde ~** et peut avoir des arguments.
3. la méthode *Print()* : elle a un type (ici **void**), un nom (ici *Print*) et éventuellement des arguments. C'est une fonction comme celles que nous avons déjà rencontrées, sauf que ne peut s'appliquer qu'à des objets de la classe.

— Dans la classe, on référence un attribut par son nom (par exemple, dans le constructeur, on affecte à l'attribut X, la valeur n).

— Dans le *main()*, le Point *P1* étant construit avec des arguments, nous appelons le constructeur que nous avons écrit. Par contre le Point *P2* est construit sans argument, on appelle alors le constructeur par défaut. Celui-ci n'étant pas explicitement écrit, C++ en fabrique un. Nous aurions pu écrire ce constructeur comme suit (faîtes-le)

```
Point()
{
     cout<<"Constructeur par défaut de Point"<<endl;
}
```

— La méthode *Print()* est alors appelée pour les objets *P1* (*P1.Print()*) et *P2*.

— Enfin, en sortant du bloc *main()*, les destructeurs de *P1* et *P2* sont appelés.

2.2.2 Prototype d'une classe

Bien que très simple, la classe Point commence à être un peu difficile à lire ; nous déclarons les membres de la classe et leur code en même temps (on parle de déclaration *inline*). On peut alors utiliser les *prototypes*, comme pour les fonctions, qui permettent de séparer déclaration et code. Pour certaines opérations et certains compilateurs, ceci est même obligatoire.

```
#include <iostream>
using namespace std;
//
// Le prototype
//
class Point
{
  public:
    double X;
    double Y;
    Point(); //noter le ;
    Point(double m, double n); //noter le ;
   ~Point(); //noter le ;
    void Print(); //noter le ;
};
//
// le code des méthodes
//
Point::Point()
```

```
{
  cout<<"Const. par défaut de Point"<<endl;
}
Point::Point(double m, double n)
{
  X=m;
  Y=n;
  cout<<"Constructeur de Point"<<endl;
}
Point::~Point()
{
  cout<<"Destructeur de Point"<<endl;
}
void Point::Print()
{
  cout<<"Point P("<<X<<","<<Y<<")"<<endl;
}
//
// le programme principale
//
int main()
{
       Point P1(4.5,2.5);
       Point P2;
       P2.X=3.1;
       P2.Y=5.1;
       P1.Print();
       P2.Print();
       return 0;
}
```

Remarques :

- Le code des méthodes est déclaré comme : **type_de_la_méthode** *Nom_de _la_classe* **: :** *nom_de_la méthode* ; Pour les constructeurs et le destructeur, il n'y a pas de type, par contre la méthode *Print()* est de type **void**. Les **: :** s'appellent "opérateur de résolution de portée" ; ils indiquent que ce qui suit (les méthodes) ne se rapporte qu'à ce qui précède (le nom de la classe).
- Les méthodes dont le code est très réduit (une voire deux lignes) peuvent rester déclarées *inline*.
- En général, on met dans un fichier la déclaration de la classe (i.e., son prototype) ; ce fichier porte le nom de la classe et a pour extension ".h" (ici, on aurait par exemple *Point.h*). La définition du code des méthodes est mis dans un fichier ayant le nom de la classe et l'extension ".cpp" (ici, *Point.cpp*).

2.2.3 Attributs private ou public

Reprenons le fichier Point.h en le modifiant comme suit :

```
#include <iostream>
using namespace std;
class Point
{
  private:
    double X;
    double Y;
  public:
    Point(){cout<<"Const. par défaut de Point"<<endl;} //declaration inline
```

```
    Point(double m, double n);
    void SetX(double n){X=n;} //declaration inline
    void SetY(double n){Y=n;} //declaration inline
    double GetX(){return X;} //declaration inline
    double GetY(){return Y;} //declaration inline
    void Print();
};
```

Cette modification fait fait que les variables X et Y ne peuvent plus être directement référencées à l'extérieur de la classe ; il faut utiliser les nouvelles méthodes *SetX()* ou *GetX()* ; par exemple, dans le *main()*, on remplacera *P2.X=3.1* par *P2.SetX(3.1)*. Dans cet exemple cela n'a pas vraiment d'intérêt ; mais vous verrez très vite que cela est une précaution qui évite beaucoup d'erreur (voir la classe Vecteur). **C'est la base de la programmation objet : on ne peut manipuler un objet qu'avec des méthodes spécifiques**.

Remarques :

- le constructeur et le destructeur doivent être **public** pour que vous puissiez créer ou détruire des objets de la classe.
- si on ne précise pas **public**, les membres sont **private** par défaut.
- La fin d'un bloc **public** (**private**) est définit par un **private :** (**public :**) ou par l'accolade de la fin de déclaration de la classe.
- On peut enchaîner plusieurs **private : ... public :** de suite dans une classe.

2.2.4 Pointeur sur un objet

Nous avons déjà utilisé les pointeurs sur des objets de base (**int**, **float**, ...). On peut aussi créer des pointeurs sur des objets d'une classe plus complexe. Considérons toujours notre classe Point et modifions la fonction *main()* :

```
int main()
{
  Point P1(4.5,2.5); // P1 est un objet Point
  Point *P2;  // P2 est un pointeur sur un objet Point
  Point *P3;  // P3 est un pointeur sur un objet Point
  P2= new Point(3.1,5.1); //----- ligne A ---
  P3= new Point();        //----- ligne B ---
  P3->SetX(1.0);          //----- ligne C ---
  P3->SetY(-2.5);
  P1.Print();             //----- ligne D ---
  P2->Print();
  P3->Print();
  Point *line;
  line=new Point[3];      //---- ligne E ---
  for(int i=0; i<3; i++)  // i++ est la notation raccourcie pour i=i+1
  {
    line[i].SetX(2.5*i+3.2);
    line[i].SetY(1.6-i);
  }
  for(i=0; i<3; i++) line[i].Print();
  delete P2;              //---- ligne F ---
  delete P3;
  delete [] line;         //---- ligne G ---
  return 0;
}
```

Remarques :

- *Ligne A : P2* est un pointeur sur un Point ; on alloue la place nécessaire à l'objet sur lequel *P2* pointe en utilisant le constructeur avec arguments.

— *Ligne B : P3* est un pointeur sur un Point ; on alloue la place nécessaire à l'objet sur lequel *P3* pointe en utilisant le constructeur par défaut.
— *Ligne C :* pour un pointeur, on accède à un membre non plus par un "point" mais par une "flèche" (composée du signe moins et du signe supérieur). Ici la méthode *SetX* est utilisée pour remplir l'attribut *X* de l'objet Point.
— *Ligne D :* par contre *P1* est un objet Point ; on utilise toujours le point pour accéder à un membre.
— *Ligne E :* La variable "*line*" est aussi un pointeur, mais cette fois sur un tableau de 3 Points ; cette ligne appelle 3 fois le constructeur *Point* (constructeur par défaut). Chaque élément du tableau *line*, `line[i]`, est un objet **Point** (et pas un `Point*`), c'est pourquoi on utilise le "." et pas la "`->`" dans l'appel des membres (ici, les méthodes *SetX* et *SetY* et *Print*)
— *Ligne F :* en quittant le programme, on ne doit pas oublier de rendre la mémoire grâce au **delete.** Noter que comme *line* est un tableau de **Point**, on est obligé de faire `delete [] line` (*Ligne* G) ; penser qu'à chaque fois que vous faîtes un **new**, il faut faire un **delete** ; **delete** appelle le destructeur de la classe.

2.3 La classe Vecteur

Dans cette partie, vous allez vous familiariser avec la notion de classe en écrivant une classe *Vecteur*.

2.3.1 Un Vecteur de taille constante

Un vecteur de R^3est définit par ses 3 composantes.

1. Vous allez construire une classe **Vecteur** ayant 1 attribut privé, un tableau *X[3]* de **double**. Vous ajouterez un constructeur par défaut qui remplit *X* avec des *0.*, un constructeur ayant comme argument un pointeur **double** ***U* qui sera un tableau de 3 valeurs. Ce constructeur copiera les valeurs *U[i]* dans *X[i]* . Vous fabriquerez un destructeur, ainsi que 3 autres méthodes, *Print()*, qui affichera les 3 coordonnées du **Vecteur**, *Get(**int** i)* qui renvoie la i$^{\text{ème}}$composante de *X* et *Set(**int** i, **double** value)* qui affecte à *X[i]* la valeur *value* .

   Vous utiliserez le *main()* suivant pour tester votre classe :

```
int main()
{
  const int n=3;
  Vecteur v;
  for(int i=0 ; i<n ; i++) v.Set(i,2.0*i);
  double u[n]={1.5,0.,2.};
  Vecteur w(u);
  v.Print();
  cout<<w.Get(2)<<endl;
  return 0;
}
```

2. Vous allez ajouter à votre classe 3 méthodes : une, *Norme()*, qui calcule la norme d'un vecteur, une autre, *PS(**Vecteur** W)* qui calcule le produit scalaire d'un **Vecteur** avec le **Vecteur** *W* et enfin une fonction *ProdVect(**Vecteur** W)* qui calcule le produit vectoriel d'un **Vecteur** avec *W*. On utilisera le *main()* suivant pour tester votre classe :

```
int main()
{
  const int n=3;
  double u[n]={1.,0.,0.};
  double v[n]={0.,1.,0.};
  double w[n]={0.,1.,1.};
  Vecteur U(u),V(v),W(w);
  cout<<"la norme de W est "<<W.Norme()<<endl;
```

```
  cout<<"U.V = "<<U.PS(V)<<endl;
  cout<<"V.W = "<<V.PS(W)<<endl;
  cout<<"V x W = "<<endl;
  V.ProdVect(W).Print();
  Vecteur Q=W.ProdVect(V); //------- ligne A------
  cout<<"W x V = "<<endl;
  Q.Print();
  return 0;
}
```

Remarque :

La *ligne A* n'est pas si triviale qu'il n'y paraît ; en effet, le résultat de *W.ProdVect(V)* est un **Vecteur** que l'on affecte au **Vecteur** *Q* : bien que nous ne l'ayons pas écrit, C++ utilise un *constructeur de recopie* en recopiant chaque attribut du **Vecteur** *W.ProdVect(V)* dans le **Vecteur** *Q* (i.e., le contenu des 3 cases de **double**). Tant que les attributs de la classe sont des variables statiques (ici un tableau statique), le constructeur de recopie automatiquement écrit par C++ convient tout à fait. Par contre, dans le cas où on aurait à faire de l'allocation dynamique (voir " *Vecteur de taille variable*"), nous devons écrire nous-même ce constructeur de recopie.

2.3.2 Vecteur de taille variable

Un vecteur de R^nest définit par ses n composantes. Vous allez modifier la classe Vecteur comme suit :

1. Remplacer l'attribut double `X[3]` par 2 attributs privés, un entier *Size* et un pointeur *X* sur un **double**. Modifier le constructeur par défaut pour que *Size* et *X* soit mis à *0*. Ajouter un constructeur ayant comme argument un entier *N* qui réserve la place nécessaire pour un **Vecteur** *V* de *N* composantes, un constructeur ayant comme arguments un entier *N* et un pointeur double *U qui sera un tableau de *N* valeurs. Ce constructeur copiera les valeurs `U[i]` dans `X[i]`. Modifiez le destructeur, ainsi que les 3 méthodes, *Print()*, *Get(int i)*, et *Set(int i, double value)* en prenant soin de vérifier que `i<Size`. Écrire une fonction *GetSize()*, qui renvoie la taille du Vecteur.

   On utilisera le *main()* suivant pour tester votre classe :

   ```
   int main()
   {
     const int n=3;
     Vecteur v(n);
     for(int i=0 ; i<n ; i++) v.Set(i,2.0*i);
     double u[n]={1.5,0.,2.};
     Vecteur w(n,u);
     v.Print();
     cout<<w.Get(2)<<endl;
     return 0;
   }
   ```

   Remarque :

   Si les attributs avaient été **public**, rien ne nous aurait empêché d'accéder à des cases non autorisées de *X*.

2. Modifier *Norme()*, *PS(Vecteur W)* et *ProdVect(Vecteur W)* . On prendra soin avant de faire le produit scalaire ou vectoriel de vérifier que les tailles des Vecteurs sont compatibles. Ajouter alors le *constructeur de recopie* [2] suivant :

   ```
   Vecteur::Vecteur(const Vecteur &U)
   {
     cout<<"const. de recopie"<<endl;
     Size=U.Size;
   ```

2. Voir ici ce qui se passe sans ce constructeur de recopie

```
  X= new double[Size];
  for(int i=0; i<Size; i++) X[i]=U.X[i];
}
```

On utilisera le *main()* suivant pour tester votre classe :

```
int main()
{
  const int n=3;
  double u[n]={1.,0.,0.};
  double v[n]={0.,1.,0.};
  double w[n]={0.,1.,1.};
  Vecteur U(n,u),V(n,v),W(n,w);
  cout<<"la norme de W est "<<W.Norme()<<endl;
  cout<<"U.V = "<<U.PS(V)<<endl;
  double ps=V.PS(W);
  cout<<"V.W = "<<ps<<endl;
  cout<<"V x W = "<<endl;
  V.ProdVect(W).Print();
  Vecteur Q=W.ProdVect(V);
  cout<<"W x V = "<<endl;
  Q.Print();
  return 0;
}
```

2.4 Surdéfinition des Opérateurs

On pourra aussi voir La surdéfinition pour plus de détails.

2.4.1 Opérateur binaire

Pour effectuer le produit scalaire entre deux vecteurs, plutôt que d'utiliser la méthode *PS(Vecteur)* ci-dessus, on aimerait utiliser la syntaxe plus agréable : `ps=V*W;` cela est possible en redéfinissant l'opérateur * (on dit que l'opérateur * est surdéfini) de la façon suivante.
Code à rajouter dans la déclaration de la classe :

```
double operator*(Vecteur U);
```

Code à rajouter dans le code des méthodes :

```
double Vecteur::operator*(Vecteur U)
{
  double ps=0;
  for(int i=0 ; i<n ; i++)
    ps+=V[i]*U.Get(i);   // ceci est equivalent à ps=ps+V[i]*U.Get(i);
  return ps;
}
```

Utilisation dans le programme principal :

```
int main()
{
  const n=3;
  double v[n]={1.,2.,3.};
  double w[n]={4.,1.,1.};
  Vecteur V(n,v),W(n,w);
  double ps=V*W;
  cout<<"V.W ="<<ps<<endl;
```

```
  return 0;
}
```

Commentaire

— La syntaxe de l'opération * est semblable à celle de la méthode *PS()*. Comme pour *PS*, Il est important de remarquer `Vecteur U` est un paramètre de l'opérateur *. Lorsque l'on écrit $ps=V*W$, l'opérateur * (ou si vous préférez la méthode *) est appelé pour l'objet V qui se trouve à gauche de * (et non W) ; à l'appel de cette opération, le vecteur U prend la valeur de W qui se trouve à droite de *.

— On dit que * est **un opérateur binaire** car lors de l'appel `ps=V*W` il y a deux entrées pour *. Il y a, à sa gauche l'objet de la classe, et à sa droite un autre objet (à priori quelconque, ici `Vecteur U`).

— Les **symboles possibles** pour un opérateur binaire sont :
`(),[ ], -> , *,/,%,+,-, <<,>>,<, <=,>,>=,==,!=,&,^, ||,&&,|,=,`
`+=,-=,*=,/=,%=,&=,^=,|=,<<=,>>=,` et la virgule `,` .

Exercice

1. Écrivez le code pour effectuer la somme (termes à termes) et la différence de deux vecteurs par la syntaxe : `w=u+v; w=u-v`.
2. Écrivez le code pour effectuer le produit et la division externe d'un réel a avec un Vecteur V (produit de chaque élément) par la syntaxe `W=V*a` ou `W=V/a` .

2.4.2 Opérateur unaire

Comme en mathématiques, on aimerait donner un sens $-V$ qui est l'opposé du vecteur V. Pour cela il faut écrire :
Code à rajouter dans la déclaration de la classe :

```
Vecteur operator-();
```

Code à rajouter dans le code des méthodes :

```
Vecteur Vecteur::operator-()
{
  return (*this)*(-1);
}
```

Utilisation dans le programme principal :

```
int main()
{
  const n=3;
  double v[n]={1.,2.,3.};
  Vecteur V(n,v),W;
  W=-W;
  W.Print();
  return 0;
}
```

Commentaires

— On dit que - est **un opérateur unaire** car lors de l'appel $W=-V$ car la seule entré de - est l'objet V de la classe qui se trouve à droite. (il n'y a pas d'entrée à gauche).

— Les **symboles possibles** pour un opérateur unaire sont :

```
+,-,++,--,!,~,*,&,new,new[],delete,(cast)
```

— **(*this)** dans la méthode signifie l'objet en cours. Ici il s'agit du vecteur V (**this** est un pointeur sur l'objet en cours).
Remarquez la façon économique de calculer le résultat : on utilise la multiplication externe déjà programmée.

Exercice

- Simplifiez le code de l'opérateur binaire w=u-v déjà écrit en utilisant l'opérateur binaire + et l'opérateur unaire - seulement.
- Simplifiez le code de l'opérateur binaire w=v/a déjà écrit en utilisant l'opérateur binaire * .

2.5 Amitié

La syntaxe permise par les méthodes ci-dessus a une petite limitation. Dans la classe Vecteur, on aimerait par exemple définir le produit externe * d'un double a avec un Vecteur V, et écrire W=a*V ;

Mais d'après la disposition des objets, il faudrait que * soit un opérateur binaire de l'objet a qui se trouve à gauche . On peut cependant associer cette opération à la classe Vecteur de l'objet V en écrivant :
Code à rajouter dans la déclaration de la classe :

```
friend Vecteur operator*(double a,Vecteur U);
```

Code à rajouter avec le code des méthodes :

```
Vecteur operator*(double a,Vecteur U)
{
   return (U*a);
}
```

Utilisation dans le programme principal :

```
int main()
{
    const n=3;
    double v[n]={1.,2.,3.};
    Vecteur V(n,v),W;
    double a=3.14;
    W=a*V;
    W.Print();
    return 0;
 }
```

Commentaire

- Le préfixe *friend* signifie que cet opérateur n'est pas exactement une méthode de la classe Vecteur, mais pas non plus indépendante puisque elle est déclarée avec la classe. Remarquez l'omission du préfixe de rattachement *Vecteur::* . On parle de **méthode ou fonction amie**.
- Les paramètres de l'opération * dans la déclaration sont *(double a,Vecteur U).* Ils se trouvent de part et d'autre de * lors de l'utilisation : *W=a*V;*
- Comme en mathématiques, on a définit a*V=V*a. Mais on pourrait écrire ce que l'on veut et ne pas respecter la commutativité.

3 le Mot de la fin

A présent, vous en savez suffisamment pour démarrer votre projet. Pour ceux que cela intéresse, le chapitre Les classes II comporte un certain nombres de compléments importants sur les classes. Dans votre projet, vous serez amenés à utiliser du graphisme graphisme ; mais pour l'instant il n'est pas utile de vous y plonger.