Les Classes II

Licence de Physique

Contents

1 Attributs statiques

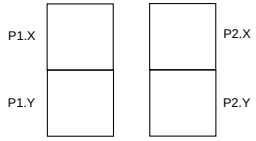
Reprenons la classe *Point* :

```
#include <iostream>
using namespace std;
class Point
{
  private:
    double X;
    double Y;
  public:
    Point(double m, double n)
    {
       X=m;
       Y=n;
       cout<<"constructeur du Point<<endl;
    }
    ~Point(){cout<<"destructeur du Point<<X<<","<<Y<<endl;}
};
```

Le fait de déclarer deux objet de la classe *Point* comme

```
Point P1(1.,1.);
Point P2(2.,2.);
```

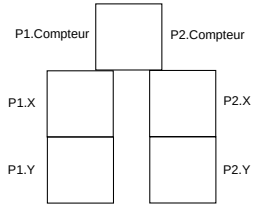

abouti à ce shéma :



Imaginer que souhaitions savoir combien de points ont été crées. Nous allons prendre un 3ème attribut de *Point*, *Compteur*. Pour que cet attribut soit **commun à tous les objets** *Point*, il faut le déclarer en **static** :

```
#include <iostream>
using namespace std;
class Point
{
  private:
   static int Compteur;
    double X;
    double Y;
  public:
    Point(double m, double n)
    {
       X=m;
       Y=n;
       Compteur++;
       cout<<"constructeur du "<<Compteur<<"eme Point<<endl;
    }
    ~Point()
    {
       cout<<"destructeur du Point<<X<<","<<Y<<endl;
       Compteur--;
       cout<<"il reste "<<Compteur<<" Point(s)"<<endl;
    }
};
int Point::Compteur=0;
```

Le shéma de la déclaration précédente pour *P1* et *P2* est cette fois,



Remarque

- Le *Compteur* **static** doit être initialisé à une valeur initiale ; c'est le rôle de la ligne *int Point::Compteur=0;* Ceci ne peut être fait qu'une fois pour toute dans le programme.

- Les variables "static" sont donc **communes à tous les objets d'une classe**.

2 Objets constants

Reprenons encore la classe *Point* :

```
#include <iostream>
using namespace std;
class Point
{
  private:
    double X;
    double Y;
  public:
    Point(double m, double n){X=m;Y=n;}
    void SetX(double n){X=n;}
    void SetY(double n){Y=n;}
    double GetX(){return X;}
    double GetY(){return Y;}
    void Print(){cout<<"Point en "<<X<<","<<Y<<endl;}
};
```

La déclaration d'un objet de cette classe se fait par

```
Point P1(1.3,2.5);
```

Si à présent nous voulons définir un **objet constant** (i.e., **qu'on ne peut pas modifié**) nous utiliserons

```
const Point P2(2.,1.);
```

Cependant, comme cet objet *P2* n'est pas modifiable, il va falloir *préciser explicitement quelles sont les méthodes* qui peuvent agir sur ce type d'*objet constant*. Ceci ce fait en modifiant les méthodes *GetX()*, *GetY()* et *Print()* (qui ne modifient pas les attributs de *Point*) par

```
double GetX() const {return X;}
double GetY() const {return Y;}
void Print() const {cout<<"Point en "<<X<<","<<Y<<endl;}
```

Avec ces définitions, les instructions suivantes sont correctes

```
P1.SetX(2.);
P1.Print();
cout<<P2.GetY()<<endl;
P2.Print();
```

Par contre

```
P2.SetX(3.);
```

produira une erreur de compilation.

3 La surdéfinition plus en détail

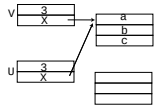
Nous avons abordé la surdéfinition des fonctions (ou méthodes de classes) dans La surdéfinition des fonctions. Nous avons vu la surdéfinition des opérateurs dans le paragraphe sur la surdéfinition et dans le chapitre sur l'amitié. Voyons encore quelques exemples qui sont souvent utiles en les illustrant par la classe Vecteur dont voici la définition :


```
class Vecteur
{
   int Size;
   double *X;
 public:
   Vecteur();
   Vecteur(int n, double *v);
   int GetSize(){return Size;}
};
Vecteur::Vecteur()
{
      Size=0;
      X=0;
}
Vecteur::Vecteur(int n, double *v)
{
   Size=n;
   X=new double[Size];
   for(int i=0 ; i<Size ; i++) X[i]=v[i];
}
```

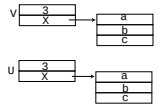
3.1 Constructeur de recopie et Opérateur =

3.1.1 Constructeur de recopie

Lors de la définition de la classe **Vecteur** pour un vecteur de taille constante nous avons écrit quelque chose du type **Vecteur V=U** où U est un **Vecteur**. La classe **Vecteur** que nous avions écrite ne faisait pas d'allocation dynamique et chaque attribut du vecteur U était automatiquement recopié dans les attributs de V. Si nous faisions la même chose sur notre nouvelle classe **Vecteur** (qui possède un pointeur "dynamique") voilà ce qui se passerait pour l'attribut X



en supposant que U soit un vecteur de R^3. En fait, nous voudrions obtenir quelque chose comme



Pour ce faire, nous devons **écrire explicitement le constructeur de recopie**. Il faut donc rajouter au prototype de la classe :

```
Vecteur(const Vecteur &U);
```

et la définition de ce **constructeur de recopie** est

```
Vecteur::Vecteur(const Vecteur &U)
{
    Size=U.Size;
```

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

3.1.2 Opérateur =

Pour les mêmes raisons, une instruction du type $V=U$ qui ne pose pas de problème dans le cas où tous les attributs ne sont pas des pointeurs dynamiques, doit faire l'objet d'une redéfinition de l'opérateur = pour ne pas conduire aux mêmes problèmes que précédemment. Le prototype de l'opérateur = sera donc (dans la classe **Vecteur**)

```
Vecteur & operator=(const Vecteur &U);
```

et le code de cette méthode sera

```
Vecteur & Vecteur::operator=(const Vecteur & U)
{
   if(this!=&U)
   {
       delete X;
       Size=U.Size;
       X=new double[Size];
       for(int i=0; i<Size;i++) X[i]=U.X[i];
   }
   return *this;
}
```

Remarques :

- Le premier & ne serre qu'à éviter d'appeler le constructeur de recopie.
- Nous devons tester si on a pas écrit $V=V$: si c'est le cas, on ne fait rien ; sinon (i.e., $V=U$) on commence par détruire l'ancien **Vecteur** V, puis on réserve l'espace nécessaire pour le pointeur X et on recopie chaque attribut de U dans ceux de V.
- Rappelons que **this** est un pointeur sur l'objet courant (si on fait $V=U$, this pointe sur V). ***this** est donc l'objet courant.

3.2 Opérateur []

Il peut être pratique d'accéder à une composante d'un **Vecteur** V en la syntaxe suivante : $V[i]$. Pour cela il suffit de surdéfinir l'opérateur **[]** en ajoutant le prototype suivant

```
double & operator[](int i);
```

au prototype de la classe et en écrivant cette méthode ainsi

```
double  & Vecteur::operator[](int i)
{
   if(i<Size) return X[i];
   else
   {
       cout<<"Index>Size"<<endl;
   }
   return 0;
}
```


Remarque :

- si on oublie le & on pourra faire $x=U[i]$; cependant, si on veut faire $U[i]=x$, il est indispensable de spécifier la transmission par référence de la valeur de retour.

3.3 Opérateur << ou >>

Il peut être pratique de pouvoir utiliser les méthodes **cout** et **cin** pour afficher ou entrer un **Vecteur**. Ces méthodes appartiennent à aux classes **ostream** et **istream**. Comme le premier argument de << ou >> sera un flot (**cout** ou **cin**), les opérateurs << et >> doivent être surdéfinis avec un lien d'amitié. Voici le code à rajouter au prototype de la classe.

```
friend ostream & operator<<(ostream & sortie, Vecteur U);
friend istream & operator>>(istream & entree, Vecteur &U);
```

Et voici le code à rajouter après les méthodes de la classe (rappelons que comme il s'agit d'amitié, ces fonctions ne sont pas des méthodes de la classe).

```
ostream & operator<<(ostream & sortie, Vecteur U)
{
   sortie<<"(";
   for(int i=0 ; i< U.Size-1 ; i++ ) sortie<<U.X[i]<<",";
   sortie<<U.X[U.Size-1]<<")";
   return sortie;
}
istream & operator>>(istream & entree, Vecteur &U)
{
  if(!U.Size)
  {
      cout<<"taille du vecteur : "<<flush ;
      entree>>U.Size;
      U.X = new double[U.Size];
  }
   cout<<"Entrer les "<<U.Size<<" composantes :"<<endl
   for(int i=0 ; i< U.Size ; i++ ) entree>>U.X[i];
   return entree;
}
```

Remarques :

- dans l'entête de `operator>>`, le Vecteur U est passé par référence car il est modifié.
- si le Vecteur U existe déjà ($Size \neq 0$), il ne faut réallouer la place pour l'attribut X de U.

4 L'héritage

L'héritage est un outils très puissant pour la réalisation et l'utilisation des classes. Nous présenterons ici que les notions de bases sur l'héritage.

Considérons un objet de la **classe A** ayant un certain nombre d'attributs et de méthodes ; maintenant, supposons que nous ayons un objet semblable à la **classe A** mais avec de petites différences (des *particularités* ou des *compléments*). On construira alors la **classe B** comme la **classe dérivée** de la **classe A** (**A** sera la **classe de base**). En faisant ceci, tous les attributs et les méthodes de **A** seront attributs et méthodes de **B**.

4.1 Principe

Reprenons l'exemple de la classe **Point** déjà utilisé :

```
class Point
{
  private:
    double X;
    double Y;
  public:
    Point(double m, double n);
    void Print();
};
Point::Point(double m, double n)
{
  X=m;
  Y=n;
  cout<<"Constructeur de Point"<<endl;
}
void Point::Print()
{
  cout<<"Point::Print : ("<<X<<","<<Y<<")"<<endl;
}
```

Supposons à présent que nous voulions construire également un point coloré : c'est un **Point** avec un attribut donnant sa couleur (représentée par un entier) ; nous allons écrire la classe **PointCol** de cette façon :

```
class PointCol : public Point
{
    int Color;
 public:
    PointCol(double x, double y,int c);
    void Print();
};
PointCol::PointCol(double x, double y,int c) :Point(x,y)
{
    Color=c;
    cout<<"Constructeur de PointCol"<<endl;
}
void PointCol::Print()
{
    cout<<"PointCol::Print :"<<endl;
    Point::Print();
    cout<<"la couleur est "<<Color<<endl;
}
```

et voici le programme principale

```
int main()
{
   cout<<"---------- Point P1 -----------"<<endl;
   Point P1(1.5,2.);
   P1.Print();
   cout<<"---------- PointCol P2 --------"<<endl;
   PointCol P2(2.1,5.3,5);
   P2.Print();
```



```
   return 0;
}
```

Remarques :

- Le constructeur de **PointCol** commence par appeller celui de **Point**. Dans notre cas, nous appelons **PointCol::PointCol()** avec 3 arguments ; les 2 premiers sont alors passés à Point::Point(). Noter la syntaxe. Si nous ne l'avions pas fait, **PointCol** aurait appelé le contructeur par défaut de **Point**.
- La méthode **PointCol::Print()** appelle explicitement la méthode **Point::Print()** pour afficher les coordonnées du point (noter l'utilisation de l'opérateur de résolution de porté ::) puis elle affiche la couleur de **PointCol**.
- Dans la classe **PointCol**, on ne peut pas agir directement sur les attributs X et Y de **Point** car ceux-ci sont **private**. S'ils avaient été **public**, ils seraient accessible de partout. Il existe un autre qualificatif, **protected**, qui permet de rendre accessible les attributs et les méthodes d'une classe de base uniquement à ces classes dérivées.

4.2 Virtuel, vous avez dit Virtuel

Reprenons l'exemple précédant en changeant le *main()* de la sorte

```
int main()
{
    cout<<"----- Point P1 ------"<<endl;
    Point P1(1.5,2.);
    P1.Print();
    cout<<"----- PointCol P2 ----"<<endl;
    PointCol P2(2.1,5.3,5);
    P2.Print();
    cout<<"----- Pointeur P3 ----"<<endl;
    Point *P3;
    cout<<"----- P3=P1 ----"<<endl;
    P3=&P1;
    P3->Print();
    P3=&P2;
    P3->Print();
    return 0;
}
```

Remarque

- Lorsque que le pointeur **Point*** pointe sur un *Point*, tout se passe comme on le souhaite : la méthode **Point::Print()** est appelée. Par contre si on fait pointer *P3* sur le **PointCol** *P2*, c'est la méthode **Point::Print()** qui est appelée et non **PointCol::Print()**. Il est possible de remédier à ce problème très simplement : il suffit de changer dans le prototype de la classe **Point** (classe mère), la déclaration de la méthode *Print()* par

  ```
  virtual void Print();
  ```

  Faîtes-le et regardez le changement.
  Quelle est la raison de ce changement ? Sans entrer dans les détails, dans le cas général, c'est le compilateur qui décide d'appeler telle méthode de telle classe. Par contre, lorsque que l'on met **virtual**, ce choix n'est pas fait par le compilateur, mais à l'exécution (on parle de **typage dynamique**).

4.3 Héritage multiple

cette partie est en cours d'élaboration....patience

5 Les patrons de fonctions et de classes

5.1 Patron de fonctions

Imaginons que nous voulions écrire une fonction *Min* qui renvoie le minimum de 2 entiers ; nous écrirons,

```
int Min(int a, int b)
{
   if (a<b) return a;
   else return b;
}
```

A présent nous aimerions avoir la même fonction *Min* pour des **float**, des **double**, ... Nous avons vu qu'il était possible, grâce au mécanisme de la surdéfinition de fonctions (ou méthodes), de définir une fonction ayant le même nom mais un type et des arguments différents. Nous pourrions alors écrire la même fonction *Min* pour des **float**, ... Cependant cela serait un peu fastidieux. Le C++ offre une possibilité très intéressante pour rendre ce travail plus simple : il s'agit des **patrons de fonctions** ("**template**" en anglais). Notre exemple précédent devient simplement

```
template <class T> T Min(T a,T b)
{
   if (a<b) return a;
   else return b;
}
```

Pour utiliser cette fonction, on fera simplement

```
int main()
{
   int a=3,b=5;
   float x=4.7, y=2.1;
   double u=2.5,v=3.1;
   cout<<"le min de ("<<a<<","<<b<<") est "<<Min(a,b)<<endl;
   cout<<"le min de ("<<x<<","<<y<<") est "<<Min(x,y)<<endl;
   cout<<"le min de ("<<u<<","<<v<<") est "<<Min(u,v)<<endl;
}
```

Remarques

- le mot **template** indique que la fonction *Min* est un **patron**.
- `<class T>` signifie que T est un "type". On peut l'utiliser autant pour typer la fonction (*T Min()*) que pour typer les arguments de celle-ci (*T a, T b*).
- On peut utiliser différents type "inconnus" comme par exemple

```
template <class T, class U> int MyFunction(T a, U b, T *X)
{
   U z;
   T *y=new T[10];
   ...
   return 0;
}
```

Dans cet exemple *MyFunction* est de type **int**, elle admet 3 arguments (un de type **T**, un de type **U** et un pointeur sur un type **T**).

5.2 Patron de classes

Cette notion de patron peut s'étendre aux classes. Reprenons la classe *Point* (encore!!!)

```
class Point
{
  private:
    double X;
    double Y;
  public:
    Point(double m, double n);
    void Print();
};
Point::Point(double m, double n)
{
  X=m;
  Y=n;
}
void Point::Print()
{
  cout<<"Point::Print : ("<<X<<","<<Y<<")"<<endl;
}
```

Si à présent nous voulons créer des *Points* d'entiers ou de **float**, nous allons utiliser les patrons comme suit :

```
template <class T> class Point
{
  private:
    T X;
    T Y;
  public:
    Point(T m, T n);
    void Print();
};
template <class T> Point<T>::Point(T m, T n)
{
  X=m;
  Y=n;
}
template <class T> void Point<T>::Print()
{
  cout<<"Point::Print : ("<<X<<","<<Y<<")"<<endl;
}
int main()
{
   Point<int> P(3,5);
   Point<double> P1(3.1,5.1);
   Point<float> P2(4.9,8.1);

   P.Print();
   P1.Print();
   P2.Print();
}
```

Remarques

- Comme pour les fonctions, avant de définir la classe *Point*, on utilise template `<class T>` pour préciser qu'il s'agit d'un patron (de classe).
- Au moment de la construction, on doit préciser le type de **Point** (par exemple `Point<int>`) par le type entre `<>`.
- Quand on définit les méthodes en dehors de la classe, on doit d'une part redéfinir le type **T** par un template, et d'autre par spécifier que la classe **Point** est de type **T** par `Point<T>::` avant de donner le nom de la méthode.

[Home|Syntaxe du C++|Fichiers|Classes I|Classes II|Graphisme]