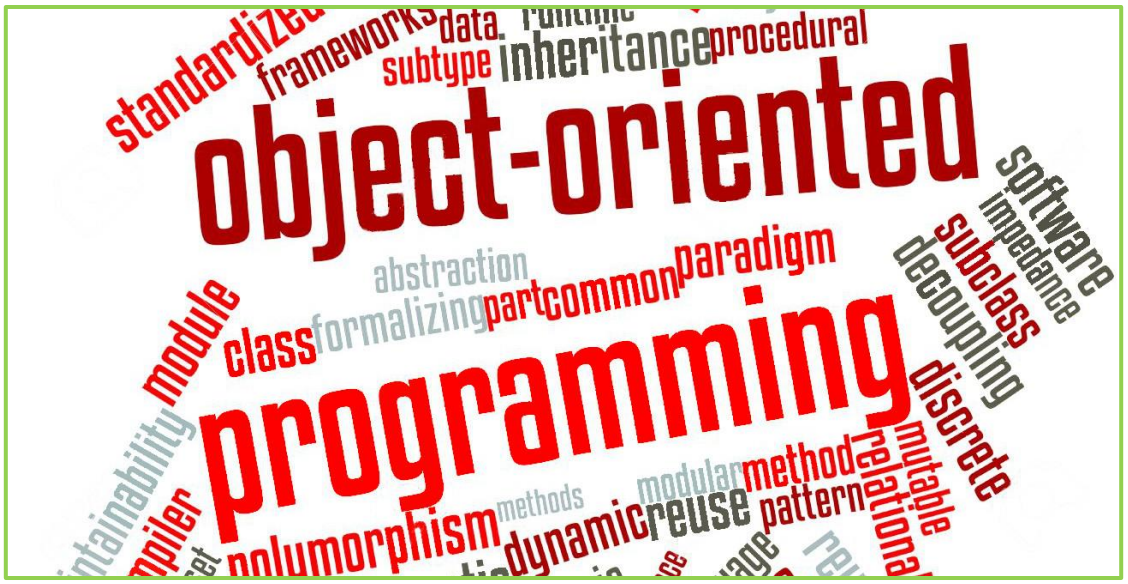
CM – Programmation C / C++

Chapitre 7 : Programmation orientée objet



I- Classes :

1) Définition d'une classe :

Dans un programme, les variables qui le composent peuvent décrire des objets réels voire même complexes. Les structures permettent de stocker plusieurs informations dans une même variable mais on peut avoir besoin que cette variable mène ses propres actions.

Une classe est une structure qui, en plus de contenir des variables, contient des fonctions qui lui sont internes. Les types et sont des exemples de classes. Une classe définit un nouveau type, comme les structures.

On qualifie les données d'une classe de données membres (ou « attributs ») et ses fonctions de fonctions membres (ou « méthodes »). Une variable dont le type est une classe est appelée un objet.

2) Syntaxe d'une classe :

```
#ifndef VECTEUR_H_INCLUDED
#define VECTEUR_H_INCLUDED

class Vecteur
{
    public :

    double x;
    double y;
    double z;

    double module();
};

#endif // VECTEUR_H_INCLUDED
```

```
#include "Vecteur.h"

double Vecteur::module()
{
    return sqrt(x*x+y*y+z*z);
}
```

Une classe, comme une structure, se définit généralement dans un header. On utilise le mot clé `class` pour la définir. Dans l'exemple donné, la classe `Vecteur` possède trois attributs et une méthode mais elle pourrait en avoir plus de chaque. Le point-virgule `;` est aussi à mettre à la fin du bloc, comme pour la structure.

Les méthodes de la classe sont généralement définies dans un fichier .cpp séparé (incluant le header). Elles fonctionnent comme des fonctions classiques. La notation signifie que c'est la méthode `module()` de la classe `Vecteur`. La méthode faisant partie de la classe, elle a accès à ses variables.

Une méthode peut être définie dans la définition de la classe, mais il vaut mieux les réserver pour des méthodes courtes (une ligne maximum) car les headers ne sont pas compilés comme les .cpp. Cela vaut pour tous les types de méthodes des classes.

3) Utilisation de la classe :

```
#include <iostream>

using namespace std;

#include "Vecteur.h"


int main()
{
    Vecteur A;
    A.x=1.;
    A.y=2.;
    A.z=2.;

    cout << "Son module = " << A.module() << endl;

    return 0;
}
```

Pour utiliser la classe, il faut inclure le header dans le fichier « main.cpp ». On peut déclarer les variables du type de la classe et on peut accéder aux attributs comme avec une structure. On appelle les méthodes comme on accède aux attributs (nom de la variable de classe suivi d'un point suivi de la méthode écrite comme n'importe quelle fonction).

4) Espace public / privé :

```
class Vecteur
{
    private :

    double x;
    double y;
    double z;

    public :

    double module();
};
```

Une classe permet de gérer qui peut accéder aux attributs et aux méthodes. Pour cela, on utilise les mots clés **`private`** (zone privée), **`protected`** (zone protégée) et **`public`** (zone publique) :

- Dans la zone privée, seuls la classe et ses amis peuvent y accéder.
- Dans la zone protégée, les héritiers de la classe (aussi appelés classes dérivées) peuvent aussi y accéder.
- Dans la zone publique, tout le monde peut y accéder.
- Par défaut, les attributs et méthodes sont privés.

Une zone est définie à partir du mot clé et jusqu'à ce qu'un prochain mot clé soit écrit ou que l'on atteigne la fin de la classe.

```
class Vecteur
{
    private :

    double x;
    double y;
    double z;

    public :

    Vecteur();
    Vecteur(double a,double b,double c);
    ~Vecteur();

    double GetX() (return x;)
    double GetY() (return y;)
    double GetZ() (return z;)

    void SetX(double a) (x=a;)
    void SetY(double a) (y=a;)
    void SetZ(double a) (z=a;)

    double module();
};
```

Dans cet exemple, la zone privé contient les attributs permettant de décrire un vecteur, tandis que la zone publique définit l'interface de la classe, c'est-à-dire l'ensemble des actions qu'un utilisateur peut faire avec elle (contient les méthodes).

Pour récupérer ou modifier la valeur d'un attribut privé, on utilise des méthodes appelées « accesseurs » (« getters ») et « mutateurs » (« setters »). On les nomme respectivement `GetAttribut()` ou `SetAttribut(…)` par convention. L'intérêt est de pouvoir contrôler l'accès aux attributs, par exemple en vérifiant que leurs valeurs soient correctes.

5) *Constructeur :*

Un constructeur est une méthode de classe appelée automatiquement lors de la déclaration d'un objet. Il sert à initialiser les attributs de la classe.

```
Vecteur(double a,double b,double c)
{
    x=a;
    y=b;
    z=c;
}
```

Le nom du constructeur est le nom de la classe. Il ne renvoie absolument rien, pas même un **`void`**. Dans cet exemple, l'utilisateur de la classe ne peut plus déclarer un objet de type `Vecteur` sans l'initialiser. On peut construire différents constructeurs pour une même classe.

```
Vecteur::Vecteur()
{
    x=0;
    y=0;
    z=0;
}

Vecteur::Vecteur(double a,double b)
{
    x=a;
    y=b;
    z=0;
}

Vecteur::Vecteur(double a,double b,double c)
{
    x=a;
    y=b;
    z=c;
}
```

Le premier constructeur est un constructeur par défaut, sans paramètres. Le second possède deux paramètres et permet d'initialiser deux attributs de la classe aux valeurs des paramètres (le dernier attribut reste nul). Le troisième possède trois paramètres et permet d'initialiser tous les attributs de la classe.

6) *Destructeur :*

Le destructeur est une méthode appelée automatiquement alors de la destruction d'un objet. Il sert à détruire proprement tous les attributs de la classe, de fermer un fichier, etc.

```
Vecteur::~Vecteur()
{

}
```

Le nom du destructeur commence par un `~` suivi du nom de la classe et de deux parenthèses. Comme le constructeur, il ne renvoie rien.

Il faudra toujours définir un destructeur si un des attributs de la classe est un pointeur (afin de supprimer l'allocution dynamique).

7) *Tableau et pointeur de classe :*

La définition d'une classe ou d'une structure entraîne la création d'un nouveau type. Comme pour n'importe quel autre type, on peut en faire des tableaux et définir des pointeurs de ce type.

```
#include <iostream>

using namespace std;

#include "Vecteur.h"


int main()
{
    Vecteur U[5];
    Vecteur *P;
    P=new Vecteur;

    double m=(*P).module;
    m=P->module();

    delete P;

    return 0;
}
```

Ici, on déclare `U` qui est un tableau de cinq vecteurs et `P` qui est un pointeur vers un `Vecteur`. On crée dynamiquement un objet de type `Vecteur` nommé `m` où `P` est son adresse mémoire et `*P` est le vecteur lui-même. On appelle ensuite la méthode `module()` qui agit sur `*P`. La ligne suivante `m=P->module();` est équivalente à la ligne précédente `double m=(*P).module();`. Avant de terminer le programme, on détruit l'objet dynamique.

8) Pointeur « this » :

Chaque classe possède, par défaut, un pointeur nommé **this** qui pointe l'objet sur lequel on applique la fonction.

```
double Vecteur::module()
{
    return sqrt(x*x+y*y+z*z);
}
```

```
double Vecteur::module(Vecteur *this)
{
    return sqrt(this->x*this->x+this->y*this->y+this->z*this->z);
}
```

À gauche, c'est le code écrit normalement par l'utilisateur. À droite, c'est la version traduite par le compilateur. À chaque fois que le pointeur **this** pointe un attribut, il récupère son adresse mémoire comme n'importe quel pointeur classique.

II- Surcharge d'opérateurs pour les classes :

Les opérateurs arithmétiques et logiques peuvent être surchargés directement à l'intérieur d'une classe. Dans ce cas, il s'agit toujours d'un opérateur entre la classe (opérande de gauche) et un autre type (opérande de droite).

```
MaClasse C1;
int n=2;
MaClasse C2=C1*n

MaClasse C2=C1.operator*(n);
```

La dernière ligne du premier paragraphe et la ligne seule sont équivalentes. On peut alors écrire la syntaxe générale pour un opérateur X :

```
type operatorX(type droite)
{
...
}
```

Les surcharges d'opérateurs s'écrivent comme n'importe quelle méthode, il faut donc les prototyper dans le header en les mettant dans la zone publique de la classe, puis l'écrire complètement dans le fichier .cpp.

1) Surcharge des opérateurs == et != :

```
bool Vecteur::operator==(Vecteur droite)
{
    if(droite.x==x && droite.y==y && droite.z==z)
        return true;
    else return false;
}

bool Vecteur::operator!=(Vecteur droite)
{
    if(droite==*this)
        return false;
    else return true;
}
```

Les surcharges d'opérateurs == et != sont des fonctions renvoyant un **bool** et prenant en paramètre un objet de la classe. Pour l'opérateur ==, si les attributs du vecteur en paramètre sont identiques aux attributs de l'opérande de gauche, la fonction renvoie **true**, sinon elle renvoie **false**.

Pour l'opérateur !=, puisque l'opérateur == est défini dans la classe, on peut simplifier l'écriture en comparant le vecteur en paramètre avec la valeur pointée par le pointeur **this**. Si elles sont identiques, la fonction renvoie **false**, sinon elle renvoie **true**. Dans le cas où l'opérateur == n'est pas défini, on écrit la surcharge de l'opérateur != de la même manière que pour l'opérateur ==, excepté que l'on change les == pour des != dans le **if** et que les valeurs retournées sont inversées.

2) Surcharge des opérateurs + et += :

```
Vecteur Vecteur::operator+(const Vecteur& droite)
{
    Vecteur S;
    S.x=x+droite.x;
    S.y=y+droite.y;
    S.z=z+droite.z;
    return S;
}

void Vecteur::operator+=(const Vecteur& droite)
{
    x=x+droite.x;
    y=y+droite.y;
    z=z+droite.z;
}

Vecteur& Vecteur::operator+=(const Vecteur& droite)
{
    x=x+droite.x;
    y=y+droite.y;
    z=z+droite.z;
    return this;
}
```

La surcharge de l'opérateur + est une fonction retournant un += et prenant en paramètre un objet de la classe (on le passe par référence et on ne le modifie pas avec le mot clé). On déclare un nouveau vecteur nommé S et on initialise chaque attribut de S comme l'addition de l'attribut de l'opérande de gauche avec celui équivalent de l'opérande de droite. La fonction retourne ensuite S.

La surcharge de l'opérateur += est une fonction ne retournant rien et prenant le même paramètre que pour +. On affecte aux attributs de l'opérande de gauche leurs valeurs respectives additionnées à celles de l'opérande de droite.

Cependant, cette définition de la surcharge de l'opérateur `+=` ne permet pas d'enchaîner plus fois la même opération sur la même ligne (par exemple, deux vecteurs `A` et `B` s'additionnent avec un troisième vecteur `C` avec l'opération `A+=B+=C;`, mais ça ne marche pas). Il faut alors modifier sa définition en changeant son type renvoyé de **`void`** à celui de la classe puis en renvoyant le pointeur **`this`** (on renvoie l'adresse de l'objet de classe pour aller plus vite).

3) *Surcharge des opérateurs d'entrée / sortie :*

```
Vecteur A;
cout << A;

Vecteur A;
cin >> A;
```

On aimerait pouvoir directement utiliser les flux d'entrée et de sortie pour afficher et entrer des informations dans une classe (problème similaire aux structures). Or, les opérateurs `>>` et `<<` sont appelés respectivement par **`istream&`** et `ostream&`, non par l'objet de classe, donc il faudra surcharger ces opérateurs à l'extérieur de la classe.

```
ostream& operator<<(ostream& flux,const Vecteur& v)
{
    flux << v.x << " " << v.y << " " << v.z;
    return flux;
}

istream& operator>>(istream& flux, Vecteur& v)
{
    flux >> v.x >> v.y >> v.z;
    return flux;
}
```

Les surcharges de ces opérateurs sont simples à écrire : la variable de type du flux choisi renverra les attributs que l'on veut afficher ou entrer de la classe. Le problème est que ces surcharges d'opérateurs ne peuvent pas accéder aux attributs privés d'une classe car elles ne sont pas des méthodes de cette dernière. Afin de régler cette situation, on déclare ces fonctions comme « amies » de la classe en les prototypant dans le header et en les précédent du mot clé **`friend`**.

```
friend ostream& operator<<(ostream& flux,const Vecteur& v);
friend istream& operator>>(istream& flux, Vecteur& v);
```

4) Surcharge de l'opérateur = :

```
A=B

A.x=B.x;
A.y=B.y;
A.z=B.z;
```

L'opérateur = est défini par défaut, on peut donc l'utiliser directement pour une classe donnée même si il n'est pas défini. Lorsque l'on écrit `A=B`, avec `A` et `B` deux objets d'une même classe, cela veut dire que les attributs de `B` sont copiés dans les attributs de `A`.

Cependant, il y a un problème si l'un des attributs de la classe est un pointeur.

```
class VectGeneral
{
    int n;
    double *elements;
    ...
};
```

```
VectGeneral A;
VectGeneral B;
A=B;
```

Si on écrit `A=B`, on aura `A.elements=B.elements`, donc les deux pointeurs de `A` et `B` vont pointer sur le même tableau sans que son contenu ne soit copié. De plus, si on détruit `A`, on détruit `B`.

```
VectGeneral& VectGeneral::operator=(const VectGeneral& droite)
{
    if(&droite==this)
        return *this;
    if(elements!=NULL)
        delete[] elements;

    n=droite.n;

    for(int i=0;i<n;i++)
        elements[i]=droite.elements[i];

    return *this;
};
```

La surcharge de l'opérateur = renvoie un objet de la classe et prend en paramètre un objet de la classe qui ne sera pas modifié. Si les deux objets ont la même adresse, on renvoie la valeur pointée par **`this`**. Sinon, si l'objet de gauche existait, il faut le détruire pour copier à sa place celui de droite. On crée ensuite dans l'objet de gauche un tableau de même taille que celui de droite et on y copie toutes les valeurs.

III- Constructeur par copie et variable de classe :

1) *Constructeur par copie :*

Le constructeur par copie est un constructeur particulier que l'on appelle dans trois cas distincts :

- On appelle une fonction avec un paramètre du type de la classe qui utilise un passage de paramètre par valeur (pour copier le paramètre).
- Une fonction renvoie un objet du type de la classe.
- On créé une variable du type de la classe en l'initialisant en même temps.

Le constructeur par copie est surchargé si au moins un des attributs de la classe est un pointeur, pour des raisons identiques à la surcharge de l'opérateur `=`.

```
VectGeneral::VectGeneral(const VectGeneral& droite)
{
    n=droite.n;
    elements=new double[n];

    for(int i=0;i<n;i++)
        elements[i]=droite.elements[i];
}
```

Il est obligatoire d'utiliser le passage par référence. Le constructeur qui définit comment on copie ne peut pas utiliser une copie comme paramètre, mais l'objet lui-même.

2) *Variable de classe :*

Une variable de classe est une variable partagée par tous les objets du type de la classe. Elle existe dans le programme même si aucun objet de la classe n'a été créé. On dit que cette variable est « statique ».

```
class Beatles;
{
    public :

    static int Nombre;
    ...
};
```

```
int Beatles::nombre=4;
```

```
if(date>08121980)
    Beatles::nombre=3;
```

Dans la définition de la classe, on signale que la variable est statique avec le mot clé **`static`** qui précède son type. Dans le fichier .cpp, on définit la valeur de cette variable avec une écriture similaire à celle de la déclaration d'une méthode. Dans le `main()`, on peut accéder à la variable statique en écrivant le nom de la classe suivi de deux doubles points `::` et du nom de la variable.

Une variable statique peut être privée. Par exemple, dans la classe **`string`**, la variable **`string`**`::npos` est une variable statique mais accessible uniquement par la classe elle-même.