Chapitre VI: Notions d'Encapsulation / Constructeurs et Destructeurs

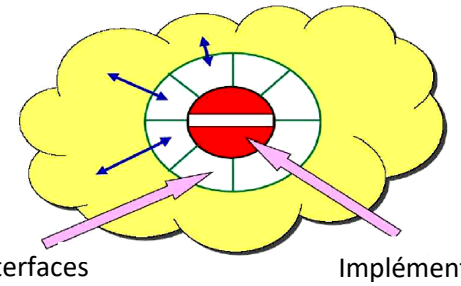
VI.1 Notion d'encapsulation

VI.1.1 Définition

L'encapsulation ou masquage d'information consiste à séparer les aspects externes d'un objet accessibles pour les autres, qu'on désigne par **publique**, des détails d'implémentation interne, rendus invisibles aux autres, qu'on désigne par **privé**.

- Technique qui isole l'aspect externe de l'objet de son aspect interne.
- Les données et les méthodes sont rassemblées ensemble. Elles sont cachées et protégées.

Ainsi **l'implémentation** d'un objet peut être modifiée sans affecter les **applications** qui emploient cet objet. Voici une figue caractéristique [11]:



VI.1.2 Notion de visibilité

La visibilité d'une caractéristique détermine si d'autres classes peuvent l'utiliser directement. On utilise 3 niveaux de visibilité:

+ (Public): Les données et fonctions membres sont accessibles dans toute l'application.

 # (Protected): Les données et fonctions membres ne sont accessibles que par les fonctions membres de la classe et de toutes classes dérivées (voir héritage).

- (Private): Les données et fonctions membres ne sont accessibles que par les fonctions membres de la classe. Par défaut les **membres sont privés.**

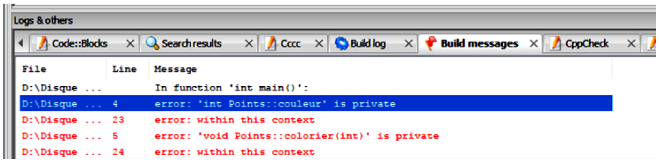
Exemple:

```
#include <iostream>
using namespace std ;
class Points
{  int couleur; //membre privé
   void colorier(int pcouleur)
   {  couleur = pcouleur; }
 public :
   int x, y;
   void afficher()
   {  cout <<this->x<<','<<this->y<<endl;} //Notation Inutile avec this
   void placer (int x , int y)
   {  this-> x = x;
      this->y = y;
  }
};
```

Programme principal:

```
#include <iostream>
#include "Points.cpp"
using namespace std ;
void main()
{   Points *p=new Points;
   Points *p1=new Points;
   p1->x=1;
   p1->y=15;
   p1->couleur=1;  // erreur de compilation car couleur est un membre privé
   p1->colorier(10);  // erreur de compilation car colorier() est une fonction privée
}
```

✍ **Messages d'erreurs:**



✍ **Remarque:** Les mots clés ***public***, ***private*** et ***protected*** peuvent apparaître à plusieurs reprises dans la déclaration d'une même classe.

VI.1.3 Méthodes d'accès et méthodes de modifications [7]

- **Les Accesseurs**: ***get<Nom Attribut>*** méthode de lecture d'un attribut privé.
- **Les Modificateurs(ou mutateurs)**: ***set<NomAttribut>*** méthode de modification d'un attribut.

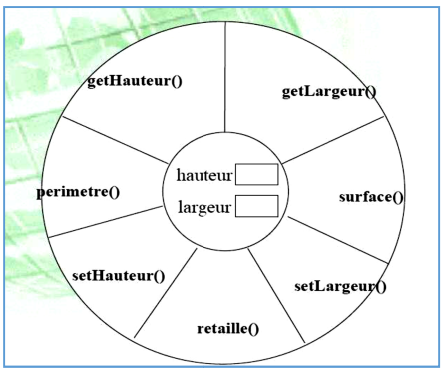
Exemple:

```
class Rectangle
{public:
   int Perimetre()
   { return 2*(largeur+hauteur); }

   int surface()
   { return largeur*hauteur; }
////////////// Les ACCESSEURS
   int getLargeur()
   { return largeur; }

   int getHauteur()
   { return hauteur; }
////////////// Les MODIFICATEURS
   void setLargeur(int nouvelleLargeur)
   { largeur=nouvelleLargeur; }

   void setHauteur(int nouvelleHauteur)
   { hauteur=nouvelleHauteur; }
private :
   int largeur;
   int hauteur;
};
```



VI.2 Initialisation et constructeur

VI.2.1 Initialisation

L'initialisation permet d'affecter des valeurs à des variables lors de leur déclaration. Comment pouvons-nous initialiser un objet? Quel mécanisme offre la programmation orientée objet pour l'initialisation des objets.

Initialisation : Affectation de valeur lors de la déclaration

Exemple:

```
class compte
{ private:
   int numero; //numéro du compte
   float solde ;
 public :
   void init(int n)
   { numero=n ;
     solde=0.f ;
   }
   void consulter()
   {  cout <<"Le compte numero:"<<numero ;
      cout <<"\n a le solde : "<<solde ;
   }
};
```

Nous voulons initialiser un objet compte b :

- **1ère solution:**

```
compte b;
b.init(100);
```

  - Correcte mais nous n'avons pas fait d'initialisation, en effet, l'affectation est faite après la déclaration. En plus nous pouvons avoir des comptes qui sont manipulés sans avoir un numéro de compte parce que nous pouvons déclarer des objets compte sans être obligé d'appeler init.

- **2ème solution:**

```
compte b={100,0.f} ;
```

  - Correcte mais numéro et solde sont privés donc nous ne pouvons pas les accéder. En plus même si dans une classe tous ses attributs sont publiques, il faut tous les initialiser et dans l'ordre de leur apparition dans la classe, ce qui n'est pas aisée.

⇒ Vous remarquez donc que pour initialiser un objet nous avons besoin d'un nouveau mécanisme: **Constructeur.**

VI.2.2 Constructeurs

Un constructeur est une fonction membre spéciale dont la tâche est d'initialiser l'objet; elle porte le même nom que la classe, elle n'a pas de résultat, et elle peut avoir 0 ou plusieurs paramètres.
Elle est appelée lors de la déclaration de l'objet d'une façon implicite.

Exemple :

```
class Cpoint
{   float abscisse;
    float ordonne;

  public:
    Cpoint (float x =0.f, float y =0.f)
    { abscisse = x;
      Ordonne = y;
    }
};
void main()
{    Cpoint p1(5,3);  //p1 a les coordonnées (5,3)
     Cpoint p2(15);  //p2 a les coordonnées (15,0)
     Cpoint p3;      //p3 a les coordonnées (0,0)
}
```

VI.2.3. Quelques règles sur les constructeurs

Un constructeur doit s'en tenir aux règles suivantes:

- Un constructeur doit porter le même nom que sa classe.
- Un constructeur ne peut pas être défini avec une valeur de retour même pas void.
- Un constructeur sans paramètres est un constructeur par défaut.
- Un constructeur dont tous les paramètres ont des paramètres par défaut est un constructeur par défaut.
- Un constructeur doit être public.

VI.2.4 Surcharge des noms de constructeurs [7]

Une classe peut comporter autant de constructeurs que nécessaire.

Exemple:

```
class compte
{   int numero; //numéro du compte
     float solde ;
 public :
   compte(int n) ;
   compte(int n,float montant) ;
   compte() ;
   void consulter() ;
   void debiter(float montant) ;
   void crediter(float montant) ;
};
void main()
{ compte b (1000);         // numéro=1000 et solde=0
  compte c(1500,200);  //numéro =1500 et solde=200
}
```

VI.2.5 Constructeur et tableau d'objet:

Si nous déclarons un tableau d'objets, le constructeur par défaut sera appelé autant de fois que la taille du tableau.

Exemple:

```
compte C[5];    //le constructeur sera appelé 5 fois
compte *pc=new compte[10];    //le constructeur sera appelé 10 fois
```

VI.3 Constructeur par recopie

Reprenons notre classe *Point*. Il apparaît qu'il pourrait être intéressant de réaliser un constructeur à partir d'un point! C'est le constructeur **par recopie**: c'est un constructeur ayant un seul paramètre objet de la même classe. Ce constructeur initialise un objet à partir du contenu des attributs du paramètre [7].

Exemple:

```
class Point       // Définition de la classe Point
{   int x;
    int y;
  public :
   Point()
    { x = -1;  Y = -1;  }
   Point(int a, int b)
   {  x = a;  y = b;   }
//Constructeur par recopie
   Point(const Point &pt)
    {  x = pt.x;   y = pt.y;
    }
... // D'éventuelles autres méthodes
};
...
Point pt(1,2);
Point pt2(pt);   //Construction par recopie de Point
...
```

⚠ C++ oblige un passage par référence dans le cas d'un constructeur par recopie. Ceci vient du fait qu'il faut réellement utiliser l'objet lui-même, et non une copie.

Dans le cas d'un objet qui comporte un pointeur, lorsqu'on veut recopier un objet, on ne recopie pas ce qui est pointé, mais l'adresse du pointeur seulement. Du coup, on se retrouve avec deux objets différents, mais qui possèdent une donnée qui pointe vers la même chose !

Exemple:

```
#include <string.h>
class PointNomme
{ public:
    PointNomme(int a, int b, char *s="")
    {   x = a;
        y = b;
        label=new char[strlen(s)+1];
        strcpy(label, s);
    }

   int x, y;
   char *label;
};
```

Regardons le Programme principal:

```
#include <iostream>
#include "PointNomme.cpp"
using namespace std ;
void main()
{
  PointNomme *a= new PointNomme(1,1);
  PointNomme *b=a;     //a et b partagent les mêmes valeurs!
  b->label="Bonjour";    //regardez cette affectation!
  a->label="Amine";
  cout <<"La chaine de a est changé mais la chaine de b vaut aussi :"<<b->label<<endl;
}
```

✍ On remarque que nous avons effectué l'affectation avant de changer l'attribut **label** de a et pourtant **b->label** a aussi changé!

Exécution :

```
La chaine de a vaut Amine et la chaine de b vaut aussi :Amine

Process returned 0 (0x0)   execution time : 2.417 s
Press any key to continue.
```

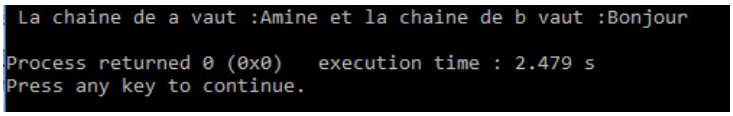
Cependant si on cherche à faire la duplication de a dans le but de gérer séparément les objets « égaux » pointés par a et b:

```
#include <string.h>
class PointNomme
{  public:
     PointNomme(int a, int b, char *s="")
     { x=a; y=b;
       label=new char[strlen(s)+1];
       strcpy(label,s);
     }
    PointNomme(const PointNomme &p) // Constructeur à partir d'un objet existant!
    { x=p.x;
      y=p.y;
      label=new char[strlen(p.label)+1];
      strcpy(label,p.label);
    }
   int x,y;
   char *label;
};
```

```
#include <iostream>
#include "PointNomme.cpp"
using namespace std ;

void main()
{  PointNomme *a= new PointNomme(1,1);
  PointNomme *b=new PointNomme(*a);
  b->label="Bonjour";
  a->label="Amine";
 cout<<" La chaine de a vaut :"<<a->label<<" et la chaine de b vaut :"<<b->label <<endl;
}
```

Résultat de l'exécution:



VI.4 Construction des objets membres

Ici on parle **d'objets membres**: lorsqu'un attribut de la classe est lui même de type une autre classe. L'initialisation d'un objet de la classe nécessite alors l'initialisation de ses objets membres. Si les objets membres n'ont que des constructeurs par défaut, le problème ne se pose pas!

Sinon voici la syntaxe :

```
NomClasse(paramètres):membre1(paramètres),membre2(paramètres),…
    {
     Corps du constructeur de NomDeLaClasse
    }
```

Voici un exemple : On considère la classe **Point** et la classe **Segment** formée par deux objets membres origine de classe Point et **extremite** de classe Point. L'appel des constructeurs s'effectue de la façon suivante :

```
class Point
{   public:
     Point(int px,int py)
     { x=px; y=py; }
  private:
     int x;
     int y;
};
class Segment
{    Point origine; //Objet membre
    Point extremite; //Objet membre
    int epaisseur;
 public:
    Segment(int ox,int oy,int ex,int ey,int ep): origine(ox,oy),extremite(ex,ey)
    { epaisseur=ep; }
};
```

IV.5 Destructeurs

Tout comme il existe un constructeur, on peut spécifier un destructeur. Ce dernier est appelé lors de la destruction de l'objet, explicite ou non.

Un destructeur d'une classe donnée est une méthode exécutée **automatiquement** à chaque fois qu'une instance de la classe donnée **disparait**.

L'identificateur est celui de la classe précédé du caractère **~** :

- C'est une méthode sans type de retour ;
- C'est une méthode sans paramètre, elle **ne peut donc pas être surchargée** ;
- C'est une méthode en accès **public**.

Exemple:

```
class Point
 { double x, y ;
   int norm ;
 public :
  // Les constructeurs
   Point(double xx, double yy);
   Point(int n);
 // Le destructeur
   ~Point();
};
```

✍ Grâce à la surcharge des noms de fonctions, il peut y avoir plusieurs constructeurs, mais il ne peut y avoir qu'un seul destructeur.

Le destructeur d'une classe C est appelé **implicitement**:

- A la disparition de chaque objet de classe C.
- A la fin du main() pour toutes les variables statiques, locales au main() et les variables globales.
- A la fin du bloc dans lequel la variable automatique C est déclarée.
- A la fin d'une fonction ayant un argument de classe C.
- Lorsqu'une instance de classe C est détruite par **DELETE**.
- Lorsqu'un objet qui contient un attribut de type classe C est détruit.
- Lorsqu'un objet d'une classe dérivée de C est détruit.

Exemple :

Le programme suivant illustre quelques cas:

```
#include <iostream>
using namespace std ;
class test
{   int attr;
  public:
    test()
    { cout<<"----- contructeur par defaut!"<<endl; }

    ~test()
    { cout<<"----- Le destructeur ! "<<endl; }
};

void blocLocal()
{  cout<<"BLOC LOCAL : "<<endl;    test CTlocal; }
void main()
{  cout<<"BLOC MAIN : "<<endl;
   test *CTmain = new test;
   cout<<"Appel du bloc local dans main : "<<endl;
   blocLocal ();
   cout<<"APPEL DE DELETE : "<<endl;
   delete CTmain;
}
```

Résultat de l'exécution :

```
BLOC MAIN :
----- contructeur par defaut!
Appel du bloc local dans main :
BLOC LOCAL :
----- contructeur par defaut!
----- Le destructeur !
APPEL DE DELETE :
----- Le destructeur !
```