Chapitre 4. Notion d'objet

4.1. Du langage C vers le langage C++

Le langage **C** (créé en 1972 par Dennis Ritchie) est l'un des piliers de l'informatique moderne. Utilisé pour développer des systèmes d'exploitation, des compilateurs, des pilotes matériels ou des logiciels embarqués, C offre un contrôle précis de la mémoire, une exécution très rapide, et une syntaxe simple. Cependant, malgré sa puissance, il devient difficile à utiliser lorsqu'il s'agit de développer des applications complexes nécessitant une organisation plus structurée. Pour répondre à ces besoins, le langage **C++** créé par Bjarne Stroustrup au début des années 1980. Son objectif n'était pas de remplacer C, mais d'étendre ses capacités, en particulier en introduisant la programmation orientée objet (POO). Son nom lui-même, *C++*, symbolise l'opérateur d'incrémentation en C, signifiant : une amélioration du C.

4.1.1. Motivations du passage de C vers C++

Limites de C	Solutions apportées par C++
Gestion complexe de gros projets	Abstraction : classes & objets
Pas de mécanisme de réutilisation automatique	Héritage et polymorphisme
Manipulation manuelle de la mémoire risquée	Pointeurs intelligents, RAII
Niveau bas parfois trop détaillé	Templates, Standard Library, algorithmes
Peu de vérifications à la compilation	Typage plus rigoureux, exceptions

C++ conserve la performance et la proximité matérielle de C, tout en offrant des outils modernes pour concevoir, organiser et sécuriser les programmes.

4.1.2. Ce que C++ ajoute par rapport à C

- Programmation orientée objet (POO)
- Classes, encapsulation, héritage, polymorphisme
- Surcharge d'opérateurs et de fonctions
- Templates et programmation générique
- Gestion automatique de ressources (RAII)
- Bibliothèque standard riche (STL)
- Pointeurs intelligents, lambdas et paradigme fonctionnel
- Support de la programmation parallèle et moderne (C++11 et +)

Contrairement à d'autres langages modernes (comme Java ou Python), C++ reste compatible avec le C : un programme C est presque toujours compilable en C++, ce qui permet une migration progressive des projets existants.

4.2.Classes et objets

La programmation orienté objet permet d'améliorer la maintenabilité (La robustesse, La modularité, Lisibilité).

Grace à la programmation orienté objets, on peut organiser des programmes complexes en utilisant les notions suivantes : l'encapsulation, l'abstraction, l'héritage et de polymorphisme.

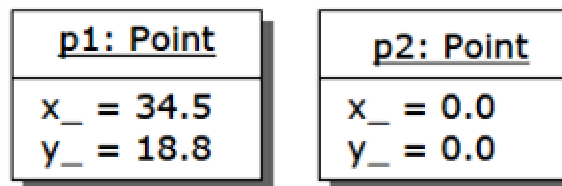
4.2.1. Objet

Un objet représente une entité individuelle et identifiable, réelle ou abstraite, avec un rôle bien défini, chaque objet peut être caractérisé par une identité, des états significatifs et par un comportement.

Objet = Etat + Comportement + Identité

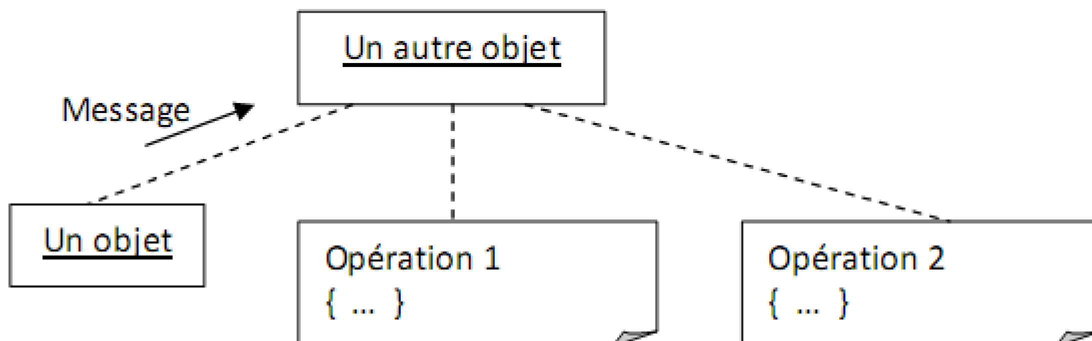
Etat : L'état d'un objet comprend les propriétés statiques (attributs) et les valeurs de ces attributs qui peuvent être statiques ou dynamiques.

Exemple : Les attributs de l'objet point en deux dimensions sont les coordonnées x et y



Comportement : Le comportement d'un objet se défini par l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés (un message = demande d'exécution d'une opération) par les autres objets.

Exemple: Un point peut être déplacé, tourné autour d'un autre point, etc.



Identité : Propriété d'un objet qui permet de le distinguer des autres objets de la même classe.

4.2.2. Notion de classe

Si des objets ont les mêmes attributs et comportements: ils sont regroupés dans une famille appelée: **Classe**. Donc, des objets similaires peuvent être informatiquement décrits par une même abstraction: une classe (Les objets appartenant à celle-ci sont les instances de cette classe) :

- Même structure de données et méthodes de traitement ;
- Valeurs différentes pour chaque objet.

a. Déclaration

Une classe contient donc :

- Des données-membres ou attributs.
- Des fonctions-membres ou méthodes.

Une classe se définit par le mot clef "**class**" suivi du nom de celle-ci (généralement commencé par une majuscule) et les données membres et les méthodes de la classe sont déclarées entre deux accolades (n'oubliez pas que la définition de la classe se termine obligatoirement par un point-virgule).

b. Syntaxe:

Le squelette d'une classe est donné par le synatxe suivant :

```
class Nom_class
{
private:
// Déclaration des attributs
// et méthodes privés
public:
// Déclaration des attributs
// et méthodes publics
        };
```

Exemple

L'exemple suivant permet de définir la classe Point aux coordonnées x et y (les attributs de classe) et les fonctions membres **afficher, changer et ajouter** (les méthodes). Le terme public signifie que tous les membres qui suivent (données comme méthodes) sont accessibles de l'extérieur de la classe.

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

class Point
{  public : // voici les attributs
     int x;
     int y;
    // voici les méthodes
    void afficher()
    { cout <<x<<','<<y<<endl; }
 void changer (int a ,int b)
    {  x=a;
       y=b;
    }
    void ajouter(int a, int b)
    {  x += a;
       y += b;
    }
};
```

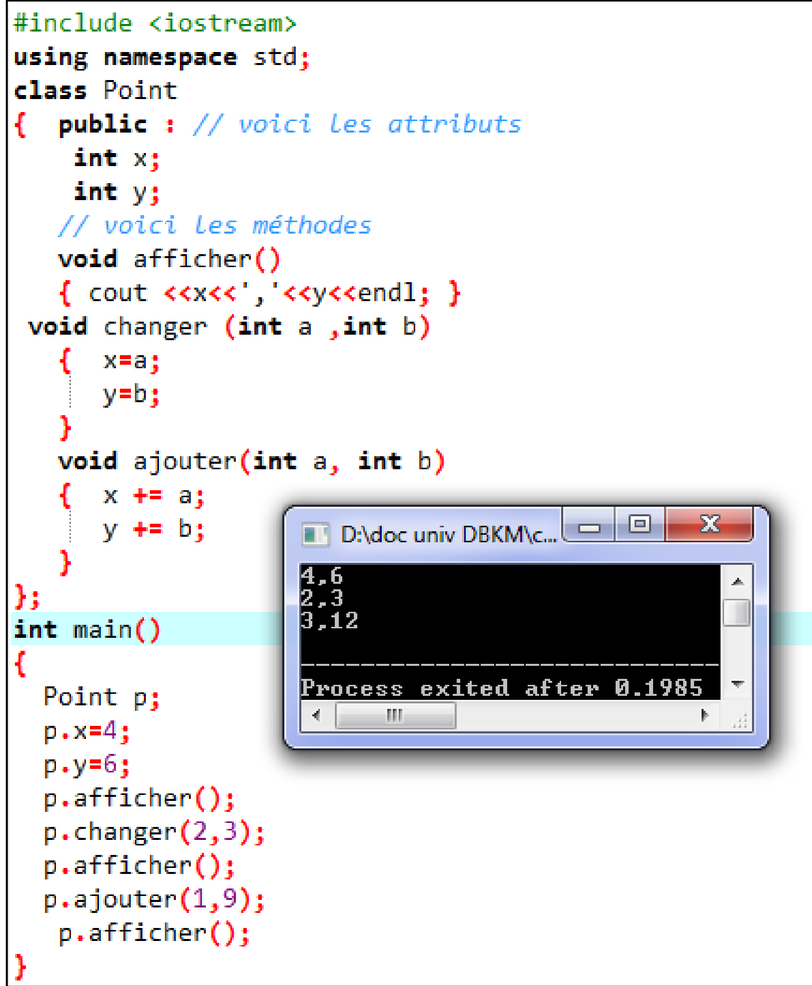
Pour déclarer un "objet" d'un type de classe donné, il suffit de précéder son nom de celui de la classe. Différents objets d'une même classe disposent des mêmes attributs et des mêmes méthodes, mais les valeurs des attributs sont différentes pour chaque objet.

Exemple :

```cpp
Point A,B;// on declare deux objets A et B  de type point
```

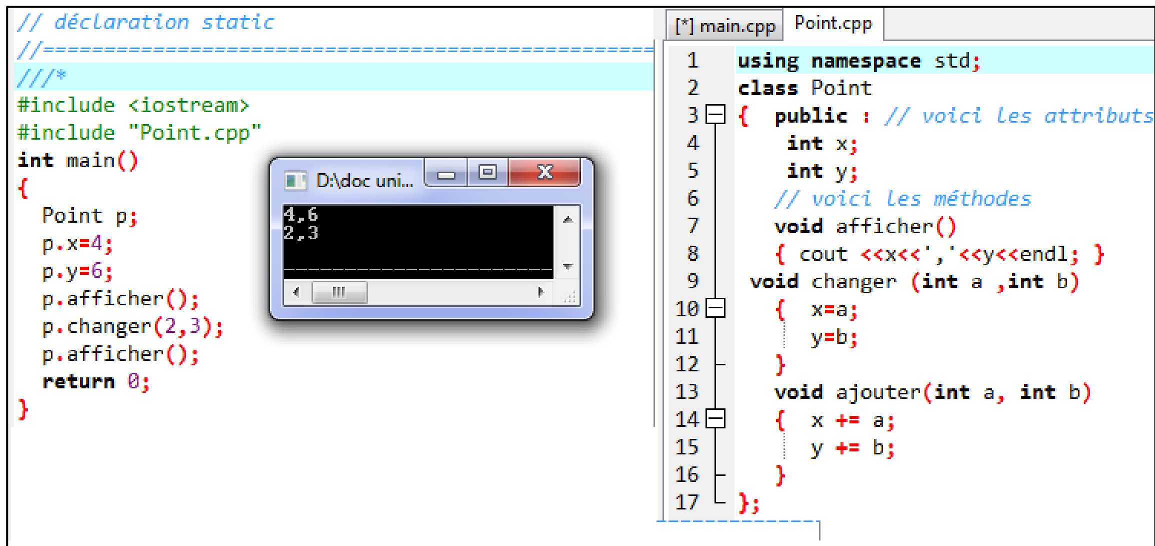
4.2.3. Accès au membre d'un objet

a. Déclaration statique



```cpp
#include <iostream>
using namespace std;
class Point
{   public : // voici les attributs
     int x;
     int y;
    // voici les méthodes
    void afficher()
    { cout <<x<<','<<y<<endl; }
  void changer (int a ,int b)
    {  x=a;
       y=b;
    }
    void ajouter(int a, int b)
    {  x += a;
       y += b;
    }
};
int main()
{
   Point p;
   p.x=4;
   p.y=6;
   p.afficher();
   p.changer(2,3);
   p.afficher();
   p.ajouter(1,9);
    p.afficher();
}
```


Ou bien

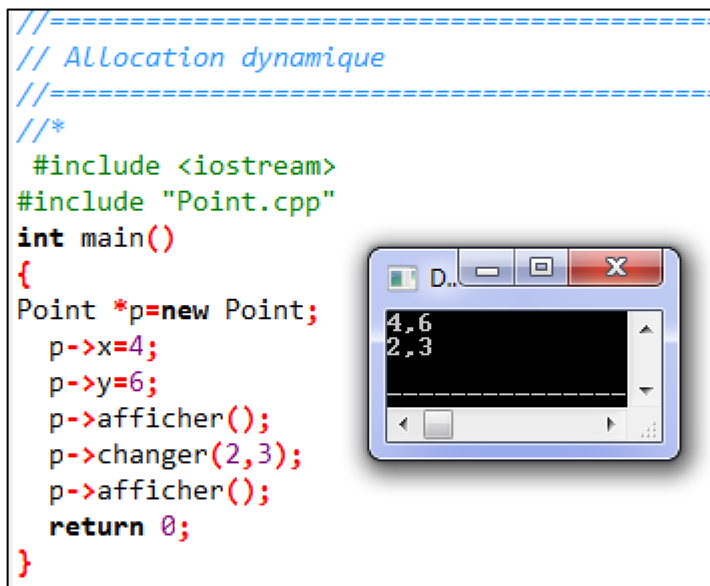


```cpp
// déclaration static
//==================================================
///*
#include <iostream>
#include "Point.cpp"
int main()
{
   Point p;
   p.x=4;
   p.y=6;
   p.afficher();
   p.changer(2,3);
   p.afficher();
   return 0;
}
```



```cpp
1  using namespace std;
2  class Point
3  {   public : // voici les attributs
4       int x;
5       int y;
6      // voici les méthodes
7      void afficher()
8      { cout <<x<<','<<y<<endl; }
9    void changer (int a ,int b)
10     {  x=a;
11        y=b;
12     }
13     void ajouter(int a, int b)
14     {  x += a;
15        y += b;
16     }
17 };
```

b. Allocation Dynamique

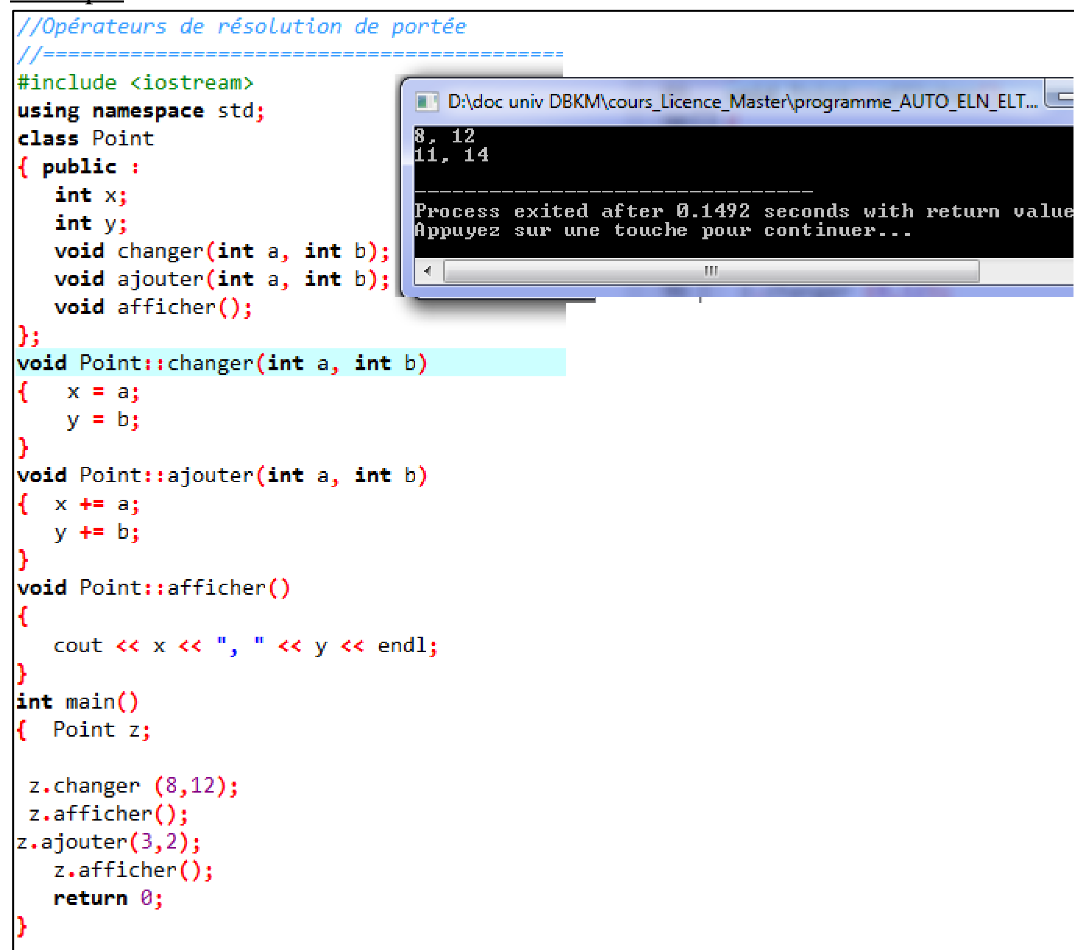


```cpp
//==========================================
// Allocation dynamique
//==========================================
//*
 #include <iostream>
#include "Point.cpp"
int main()
{
Point *p=new Point;
  p->x=4;
  p->y=6;
  p->afficher();
  p->changer(2,3);
  p->afficher();
  return 0;
}
```


c. Opérateurs de résolution de portée

Les méthodes de la classe Point sont implémentées dans la classe elle-même. Lorsque le code est plus long, cela devient assez lourd. Il est donc préférable de placer la déclaration uniquement, dans la classe.

Exemple



```cpp
//Opérateurs de résolution de portée
//=========================================
#include <iostream>
using namespace std;
class Point
{ public :
   int x;
   int y;
   void changer(int a, int b);
   void ajouter(int a, int b);
   void afficher();
};
void Point::changer(int a, int b)
{   x = a;
    y = b;
}
void Point::ajouter(int a, int b)
{  x += a;
   y += b;
}
void Point::afficher()
{
   cout << x << ", " << y << endl;
}
int main()
{  Point z;

 z.changer (8,12);
 z.afficher();
z.ajouter(3,2);
   z.afficher();
   return 0;
}
```


"Point ::" (:: opérateur de résolution de portée) signifie que la fonction est une méthode de la classe Point.

4.3. Protection et accès

La programmation en C++ repose sur la notion d'encapsulation, principe fondamental de la POO (Programmation Orientée Objet). L'encapsulation consiste **à** protéger les données d'un objet afin d'empêcher leur manipulation directe depuis l'extérieur de la classe et garantir l'intégrité des informations. Trois niveaux d'accès aux attributs et méthodes définissant les restrictions d'accès aux fonctions et variables, ceci renforce l'encapsulation ou le masquage des données. Ces trois niveaux sont définis par les mots-clés public, private ou protected.

4.3.1. Private

Une fonction private ne peut être appelée qu'à partir des fonctions membres de la classe, et qu'une donnée membre private ne peut être lue ou modifiée que par des fonctions membres de la classe.

4.3.2. Public

Pour les données et variables membres public, on peut y accéder à l'extérieur comme à l'intérieur de l'objet et il n'y a pas de restriction.

4.3.3. Protected

C'est le cas d'intermédiaire entre private et public. Les fonctions et variables membres de type protected sont accessibles par toute fonction membre de l'objet ou d'une des classes dérivées de l'objet (La notion de classe dérivée se reportera au prochaine chapitre traitant des notions d'héritage et de polymorphisme).

Chaque membre se voit définir un droit d'accès par l'un des mots-clés public, private ou protected. Seuls les membres publics sont accessibles de l'extérieur de la classe.

Exemple

```
//==================================================
//Notion de visibilité
//==================================================
//Programme principal
#include <iostream>
#include "Cordonnee.cpp"
using namespace std ;
int main()
{   Cordonnee *p=new Cordonnee;
    Cordonnee *p1=new Cordonnee;
    p1->x=13;
    p1->y=15;
    p1->couleur=1;  // erreur de compilation
    p1->colorier(10);  // erreur de compilation
    p1->afficher() ;
    return 0;
}
```

main.cpp | Cordonnee.cpp

```
1  #include <iostream>
2  using namespace std ;
3  class Cordonnee
4  {
5  int couleur; //membre privé
6     void colorier(int pcouleur)
7     {  couleur = pcouleur; }
8   public :
9     int x, y;
10    void afficher()
11    {  cout <<x<<','<<y<<endl;}
12     void placer (int x , int y)
13    {  this-> x = x;
14       this->y = y;
15    }
16 };
```

Compiler (7) | Resources | Compile Log | Debug | Find Results | Close

Line	Col	File	Message
605	0	D:\doc univ DBKM\cours_Licence_Master\programm...	In file included from D:\doc univ DBKM\cours_Licence_Master\pro
		D:\doc univ DBKM\cours_Licence_Master\programm...	In function 'int main()':
5	5	D:\doc univ DBKM\cours_Licence_Master\programme_A...	[Error] 'int Cordonnee::couleur' is private
612	9	D:\doc univ DBKM\cours_Licence_Master\programme_A...	[Error] within this context
605	0	D:\doc univ DBKM\cours_Licence_Master\programm...	In file included from D:\doc univ DBKM\cours_Licence_Master\pro
6	9	D:\doc univ DBKM\cours_Licence_Master\programme_A...	[Error] 'void Cordonnee::colorier(int)' is private
613	20	D:\doc univ DBKM\cours_Licence_Master\programme_A...	[Error] within this context

5. Variables d'instance en C++

Une variable d'instance (ou attribut d'objet) est une variable qui :

- Est déclarée à l'intérieur d'une classe,
- N'appartient pas à la classe elle-même,
- Appartient à chaque objet créé à partir de cette classe.

```
class Compte {
private:
    double solde;   // variable d'instance
};
```

solde est une variable d'instance**.** Elle appartient à chaque objet de type **Compte**, pas à la classe. Chaque objet possède sa propre copie de ces variables, contrairement aux variables statiques (qui appartiennent à la classe entière).

Exemple :

```
class Compte {
private:
    double solde;   // Variable d'instance

public:
    Compte(double s) : solde(s) {} // Constructeur

    double getSolde() const {
        return solde;
    }
};

int main() {
    Compte c1(150.0);
    Compte c2(500.0);

    cout << c1.getSolde(); // Affiche : 150
    cout << c2.getSolde(); // Affiche : 500
}
```

Chaque objet a sa propre valeur, indépendante des autres. On place les variables d'instance en private pour protéger les données sensibles et empêcher les modifications incorrectes depuis l'extérieur. On force l'accès via des méthodes contrôlées (getter, setter)

<u>Exemple :</u>

```
1  class Compte {
2  private:   //public:
3      double solde;
4
5  public:
6      void deposer(double montant) {
7          if(montant > 0) solde += montant;
8      }
9  };
10
11 int main () {
12     Compte c;
13 c.solde = -10000; // Incohérent et possible si public !
14
15
16 }
17
```

Col	File	Message
	C:\Users\AEK\Desktop\cours C++\chap4.cpp	In function 'int main()':
12	C:\Users\AEK\Desktop\cours C++\chap4.cpp	[Error] 'double Compte::solde' is private
3	C:\Users\AEK\Desktop\cours C++\chap4.cpp	[Error] within this context

6. Constructeur et Destructeur

Toute classe nécessite d'être crée puis détruite, pour cela il faut un constructeur et un destructeur. Le constructeur alloue implicitement de la mémoire et permet d'initialiser les variables internes. Le destructeur quant à lui est appelé lorsque la classe va être détruite et libère ainsi la mémoire.

6.1.Les 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.

Tout constructeur doit répondre aux caractéristiques suivantes :

-Est une fonction membre qui porte le même nom que sa classe,

-Est appelé après l'allocation de l'espace mémoire destiné à l'objet,

-Ne renvoie pas de valeur (pas même void ne doit figurer devant sa déclaration ou sa définition).

-Si aucun constructeur n'est déclaré, un constructeur par défaut, sans paramètres, est automatiquement créé.

Il existe trois types de constructeurs :

1-Constructeur par défaut (sans paramètres)

2-Constructeur d'initialisation ou paramétré (avec paramètres)

3-Constructeur par copie (reçoit en paramètre un objet)

6.1.1. **Constructeur par défaut** (le constructeur sans paramètres).

Une classe qui ne déclare aucun constructeur explicitement en possède en fait toujours un : le constructeur vide par défaut, qui ne prend aucun paramètre, il définira les variables à 0 et les chaînes à vide.

Exemple

Un constructeur vide par défaut est créé dans cette classe.

```
#include <iostream>
using namespace std;
class point
{      private :
           int x,y;
       public :
           point();
           void affiche();
};
point::point()
{
     x=0;
     y=0;
}
void point::affiche()
{
     cout<<"Le point est en position: "<<x<<","<<y<<endl;}
   int main()
{
  point p1 ;
p1.affiche();
}
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AU...
Le point est en position: 0,0
--------------------------------
Process exited after 0.5038 seconds with return value
Appuyez sur une touche pour continuer...
```

6.1.2. Constructeur d'initialisation (le constructeur paramétré)

Si l'on définit un constructeur explicite dans cette classe (i.e. la classe point), alors ce constructeur vide par défaut n'existe plus (il n'est plus créé) ; on doit le déclarer explicitement si l'on veut encore l'utiliser. On ne peut plus instancier cette classe comme précédemment. On ne peut l'instancier que de la façon suivante :

Exemple

```
#include <iostream>
using namespace std;
class point
{      private :
           int x,y;
       public :
           point(int a, int b);
           void affiche();
};
point::point(int a, int b)
{
     x=a;
     y=b;
}
void point::affiche()
{
     cout<<"Le point est en position: "<<x<<","<<y<<endl;}
 int main()
 {
    point p1(11,10) ;
 p1.affiche();
    }
```

```
D:\doc univ DBKM\cours_Licence_Maste...
Le point est en position: 11,10
--------------------------------
```

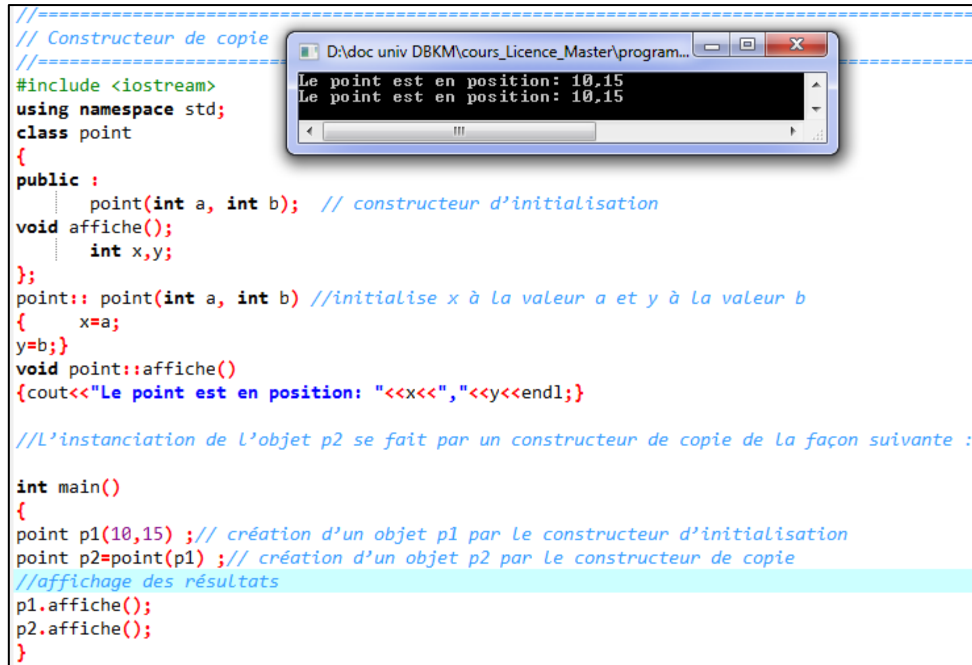
6.1.3. Constructeur de copie

Il doit effectuer une copie d'un objet du même type. Il sert à créer une copie explicitement, Il est aussi utilisé implicitement par le compilateur, comme le constructeur sans arguments, toutes les classes en ont un par défaut.

Ce constructeur par défaut recopie simplement toutes les données membres de l'objet initial transmises en arguments comme données membres du nouvel objet.

<u>Exemple :</u>

La classe point définit un constructeur de copie qui prend comme argument une instance de point. Les valeurs des propriétés de l'argument sont assignées aux propriétés de la nouvelle instance de point. Le code contient un autre constructeur de copie qui envoie les propriétés x et y de l'instance que vous voulez copier au constructeur d'instance de la classe.



```
//==========================================================================
// Constructeur de copie
//==========================================================================
#include <iostream>
using namespace std;
class point
{
public :
        point(int a, int b);  // constructeur d'initialisation
void affiche();
        int x,y;
};
point:: point(int a, int b) //initialise x à la valeur a et y à la valeur b
{     x=a;
y=b;}
void point::affiche()
{cout<<"Le point est en position: "<<x<<","<<y<<endl;}

//L'instanciation de l'objet p2 se fait par un constructeur de copie de la façon suivante :

int main()
{
point p1(10,15) ;// création d'un objet p1 par le constructeur d'initialisation
point p2=point(p1) ;// création d'un objet p2 par le constructeur de copie
//affichage des résultats
p1.affiche();
p2.affiche();
}
```


On remarque que les coordonnées du point x et y ne sont plus private, puisqu'on les modifie après les avoir construit.

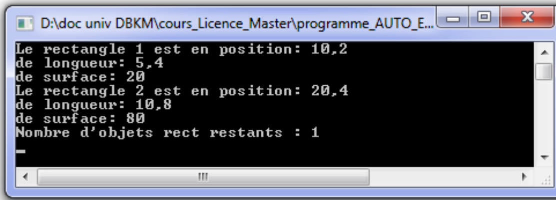
6.2. Destructeur

Le destructeur permet de détruire l'objet lorsqu'il devient inutile. Les caractéristiques d'un destructeur sont :

- Il est unique.
- Est une fonction membre portant le même nom que sa classe, précédé du symbole (~),
- Est appelé avant la libération de l'espace mémoire associé à l'objet
- Ne peut pas comporter d'arguments et il ne renvoie pas de valeur (aucune indication de type ne doit être prévue).
- Il doit restituer la place mémoire allouée dynamiquement par l'objet.

Le destructeur est une méthode particulière qui est définie implicitement pour tous les objets. Par défaut il ne fait rien.

<u>Exemple 1 :</u>




```cpp
//==========================================================================
#include <iostream>
#include <conio.h>
using namespace std ;


class rectangle
{
    private :
        int x,y;
        float lon,lar;
        static int ctr ; // compteur d'objets
    public :
        rectangle (int,int,float,float); // Constructeur
        float surface();
        void affiche(int);
        ~rectangle(); // Destructeur
};
 int rectangle::ctr = 0; // init. A 0 du nb d'objets rectangle

rectangle::rectangle(int a, int b, float c, float d) //constructeur
{
        x=a;
        y=b;
        lon=c;
        lar=d;
       ctr++;
}

float rectangle::surface()
{
    return lon*lar;
}

void rectangle::affiche(int n)
{
     cout<<"Le rectangle "<<n<<" est en position: "<<x<<","<<y<<"\n"
    <<"de longueur: "<<lon<<","<<lar<<"\n"
    <<"de surface: "<<surface()<<endl;
  }
rectangle::~rectangle() // destructeur
{
    ctr-- ;
    cout << "Nombre d'objets rect restants : " << ctr << endl ;
}

int main()
{
    rectangle rec1(10,2,5,4);   //construction d'un rectangle rec1
    rectangle rec2(20,4,10,8); //construction d'un rectangle rec2
    rec1.surface();
    rec2.surface();
   rec1.affiche(1);
   rec2.affiche(2);

    rec2.~rectangle();         //destruction du rectangle rec2
   getch();
      return 0;
}
```

6.3.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.

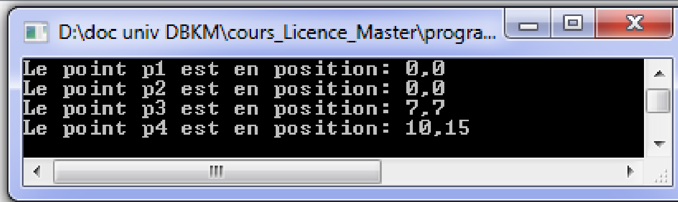
Exemple

```cpp
compte C[8];      //le constructeur sera appelé 8 fois
compte *pc=new compte[12];     //le constructeur sera appelé 12 fois
```

7. Surcharge de constructeurs

Les constructeurs peuvent être surchargés si la classe contient plus d'un constructeur.

Exemple1




```cpp
#include <iostream>
using namespace std;
class point
{
private :
    int x,y;
public :
    point();                // constructeur par défaut
    point(int a);           // constructeur d'initialisation avec un seul paramétre
    point(int a, int b);    // constructeur d'initialisation avec deux paramétres
    void affiche(int n);
};
point::point()  //initialise x à la valeur 0 et y à la valeur 0
{
    x=0;
    y=0;
}
point:: point(int a) //initialise x à la valeur a et y à la valeur a
{
    x=a;
    y=a;
}
point:: point(int a, int b) //initialise x à la valeur a et y à la valeur b
{
    x=a;
    y=b;
}
void point::affiche(int n)
{
    cout<<"Le point p"<<n<<" est en position: "<<x<<","<<y<<endl;
}
int main()
{
    point p1,p2;// appel du constructeur par défaut par les objets p1 et p2
    point p3(7) ;// appel du constructeur d'initialisation avec un seul paramétre par l'objet p3
    point p4(10,15); // appel du constructeur d'initialisation avec deux paramétres par l'objet p4
    p1.affiche(1);
    p2.affiche(2);
    p3.affiche(3);
    p4.affiche(4);
}
```

On peut utiliser un tableau d'une dimension. L'instanciation devient alors :

```cpp
point p[3]= { point(7), point(10,15) } //On crée un tableau de 3 objets de type point.
                                       //Chaque élément du tableau est un objet point.

/*On fournit deux initialisations explicites :
point(7) : constructeur avec un seul paramètre.
point(10,15) : constructeur avec deux paramètres.
Le troisième élément du tableau p[2] n'est pas initialisé explicitement ?
il sera construit automatiquement avec le constructeur par défaut. */
```

Élément	Initialisation	Type de constructeur
p[0]	point(7)	Constructeur 1 paramètre
p[1]	point(10,15)	Constructeur 2 paramètres
p[2]	rien fourni	Constructeur par défaut

Exemple 2

Utilisez la classe rectangle de l'exemple précédent., on utilisera cinq (5) constructeurs, qui sont définis comme suit :

1. Un constructeur par défaut : rectangle();
2. Un constructeur d'initialisation ou paramétré avec la longueur donnée : **rectangle(lon1)** ;
3. Un constructeur d'initialisation ou paramétré avec la largeur donnée : **rectangle(lar1)** ;
4. Un constructeur d'initialisation avec deux paramètres qui sont la longueur et la largeur : **rectangle(lon1, lar1) ;**
5. Un constructeur d'initialisation avec les position x et y initialisé ainsi que les dimensions x et y : **rectangle(x1,y1,lon1, lar1);**

```
//
//Surdéfinition ou surcharge de constructeurs  exemple 2
//===========================================================================
#include <iostream>
using namespace std;

class rectangle
{
    private :
        int x,y;
        float lon,lar;
    public :
         rectangle();
         rectangle(int);
         rectangle(int,int);
         rectangle(int,int, int);
         rectangle(int,int, int, int);
         void deplace(int dx, int dy);
         int surface();
         int circonference();
         void affiche(int n);
};
 rectangle::rectangle()
{
    x=0;
    y=0;
   lon=0;
   lar=0;
}
 rectangle::rectangle(int lon1)
{
    x=8;
    y=5;
    lon=lon1;
    lar=6;
}
 rectangle::rectangle(int lon1, int lar1)
{
    x=8;
    y=0;
    lon=lon1;
    lar=lar1;
}
 rectangle::rectangle(int x1, int y1,int lar1)
{
     x=x1;
     y=y1;
    lon=lar1;
    lar=lar1;
}
 rectangle::rectangle(int x1, int y1, int lon1, int lar1)
{
    x=x1;
    y=y1;
    lon=lon1;
```

```
D:\doc univ DBKM\cours_Licence_Master...
Le rectangle 1 est en position: 0,0
de longueur: 0,0
de surface: 0
de perimetre: 0
Le rectangle 2 est en position: 8,5
de longueur: 3,6
de surface: 18
de perimetre: 18
Le rectangle 3 est en position: 8,0
de longueur: 8,2
de surface: 16
de perimetre: 20
Le rectangle 4 est en position: 3,6
de longueur: 4,4
de surface: 16
de perimetre: 16
Le rectangle 5 est en position: 12,6
de longueur: 5,4
de surface: 20
de perimetre: 18

--------------------------------
Process exited after 0.0408 seconds with r
```

```
    lar=lar1;
}
void rectangle::deplace(int dx,int dy)
{
    x=x+dx;
    y=y+dy;}
    int rectangle::surface()
{
    return lon*lar;
}
int rectangle::circonference()
{
    return 2*(lon+lar);
}
void rectangle::affiche(int n)
{
    cout<<"Le rectangle "<<n<<" est en position: "<<x<<","<<y<<"\n"
    <<"de longueur: "<<lon<<","<<lar<<"\n"
    <<"de surface: "<<surface()<<"\n"
    <<"de perimetre: "<<circonference()<<endl;
}
int main()
{
    rectangle rec1;
    rectangle rec2(3);
    rectangle rec3(8,2);
    rectangle rec4(3,6,4);
    rectangle rec5(10,2,5,4);
        rec1.surface();
    rec1.surface();
    rec1.circonference();
   rec1.affiche(1);
      rec2.surface();
   rec2.circonference();
   rec2.affiche(2);
      rec3.surface();
   rec3.circonference();
   rec3.affiche(3);
    rec4.surface();
   rec4.circonference();
   rec4.affiche(4);
     rec5.deplace(2,4);
   rec5.surface();
   rec5.circonference();
   rec5.affiche(5);

   return 0;
}
```

8. L'opérateur this

En C++, **this** est un pointeur implicite présent dans toutes les méthodes non statiques d'une classe. Il représente **l'objet courant**, c'est-à-dire celui qui a appelé la méthode. Chaque fois qu'une méthode est exécutée, **this** pointe vers l'objet qui exécute cette méthode.

Si on est dans une classe X, alors this est un pointeur de type : **X* this;**

Cela signifie que **this** pointe vers un objet X.

Quand les paramètres d'un constructeur portent le même nom que les attributs, `this` permet d'éviter la confusion :

```cpp
class Point {
private:
    int x, y;

public:
    Point(int x, int y) {
        this->x = x;  // x à gauche = attribut, à droite = paramètre
        this->y = y;
    }
};
```

Certaines méthodes renvoient l'objet courant pour permettre le chaînage :

```cpp
class Compte {
private:
    double solde;

public:
    Compte& ajouter(double montant) {  //Compte& signifie :1.la méthode retourne une référence (un alias);
                                       //2.vers un objet de type Compte
                                       //en l'occurrence ? l'objet courant (this)
                                       //Compte& ? type de retour
                                       //ajouter ? nom de la méthode
                                       //(double montant) ? paramètre : un nombre en virgule flottante
        solde += montant;
        return *this;   // On retourne l'objet courant
    }
};

int main() {
    Compte c;
    c.ajouter(50).ajouter(30); // Chaînage de méthodes
}
```

Compte& ajouter(double montant) : Cette ligne correspond à la déclaration d'une méthode dans la classe Compte.

1) Signature complète de la méthode

```cpp
Compte& ajouter(double montant)
```

La signature se lit comme :

- Compte& → **type de retour**
- ajouter → **nom de la méthode**
- (double montant) → **paramètre** : un nombre en virgule flottante

2) Pourquoi le type de retour est Compte& ?

Compte& signifie :

- La méthode **retourne une référence** (un alias)
- Vers un objet de type Compte
- En l'occurrence → **l'objet courant (this)**

En d'autres termes, cette méthode retourne :**le même objet sur lequel elle a été appelée**, pas une copie.

3) Pourquoi ne pas retourner Compte tout court ?

❌ Mauvais : Compte ajouter(double montant)

Cela retournerait une **copie** de l'objet :

- Perte de performance
- Chaînage impossible ou inefficace
- Modifications faites sur la copie (pas sur l'original)

Donc on évite de retourner un objet **par valeur**.

Passer l'objet courant à une autre fonction

Ce code illustre l'usage de l'opérateur **this** et de la passation d'objet à une fonction **externe**.

Il montre comment une méthode d'une classe peut :

- Récupérer l'adresse de l'objet courant via this
- Passer cet objet à une fonction indépendante (non membre de la classe)

```
void afficherCompte(const Compte* c);

class Compte {
public:
    void afficher() {
        afficherCompte(this);
    }
};
```

Lorsque la méthode est déclarée const, this devient un pointeur **vers un objet constant** :

```
void afficher() const {
    // Ici, this est de type : const Compte* this
}
```

Les fonctions static n'appartiennent pas à un objet mais à la classe. Donc, il n'y a **pas d'objet courant**, donc pas de this.

```
class Test {
public:
    static void f() {
        // this; ? ERREUR
    }
};
```

<u>Exemple</u>

```cpp
//Exemple
#include <iostream>
using namespace std;

class Rectangle {
private:
    int largeur, hauteur;

public:
    Rectangle(int largeur, int hauteur) {
        this->largeur = largeur;
        this->hauteur = hauteur;
    }

    Rectangle& doubler() {
        this->largeur *= 2;
        this->hauteur *= 2;
        return *this;
    }

    void afficher() const {
        cout << "Rectangle : " << largeur << " x " << hauteur << endl;
    }
};

int main(){
    Rectangle r(5, 3);
    r.doubler().afficher(); // chaînage : doubler() retourne *this
}
```

Exercice 1

Créez une classe "cercle" avec les fonctions de membre public suivantes : positionx, positiony, area, circonference, move et rayon. Ces fonctions renvoient respectivement la position en x, la position en y, l'aire du cercle, la circonférence du cercle et le rayon du cercle. Utilisez un constructeur et un destructeur pour créer et détruire des objets cercle.

```cpp
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class circle
{
    private :
        int x,y;
        float ray;
        static int ctr ; // compteur d'objets
    public :
        circle (int,int,float); // Constructeur
        float area();
        float circonference();
        float positionx(); //position en x de l'obje
        float positiony(); //position en y de l'obje
        float rayon();          // le rayon du cercle
        ~circle();              // Destructeur
};
int circle::ctr = 0; // init. A 0 du nb d'objets cercl
circle::circle(int a, int b, float c) //Constructeur
{
    x=a;
    y=b;
    ray=c;
    ctr++;
    cout<<"constrcteur="<<ctr<<endl;
```

```
}
float circle::area()
{
    return M_PI*pow(ray,2);
}
float circle::circonference()
{
    return 2*M_PI*ray;
}
float circle::positionx()
{
    return x;
}
float circle::positiony()
{
    return y;
}
float circle::rayon()
{
    return ray;
}
circle::~circle() // destructeur
{
    ctr=ctr-1 ;
    cout << "Nombre d'objets circle restants : " << ctr << endl ;
}
int main(int argc, char* argv[])
{
    circle circle1(10,2,5); //construction d'un cercle cercle1
    circle circle2(20,4,10); //construction d'un cercle cercle2
    circle1.area();
    circle2.area();
    circle1.circonference();
    circle2.circonference();
   // affichage les elements de cercle 1
    cout<<"Le cercle1 est en position: "<<circle1.positionx()<<","<<circle1.positiony()<<"\n"
    <<"de rayon: "<<circle1.rayon()<<"\n"
    <<"de surface: "<<circle1.area()<<"\n"
    <<"de circonference: "<<circle1.circonference()<<"\n";
  // affichage les elements de cercle 2
  cout<<"Le cercle2 est en position: "<<circle2.positionx()<<","<<circle2.positiony()<<"\n"
    <<"de rayon: "<<circle2.rayon()<<"\n"
    <<"de surface: "<<circle2.area()<<"\n"
    <<"de circonference: "<<circle2.circonference()<<"\n";
    circle1.~circle(); //destruction cercle1
    circle2.~circle(); //destruction cercle2
    getch();
    return 0;
}
```

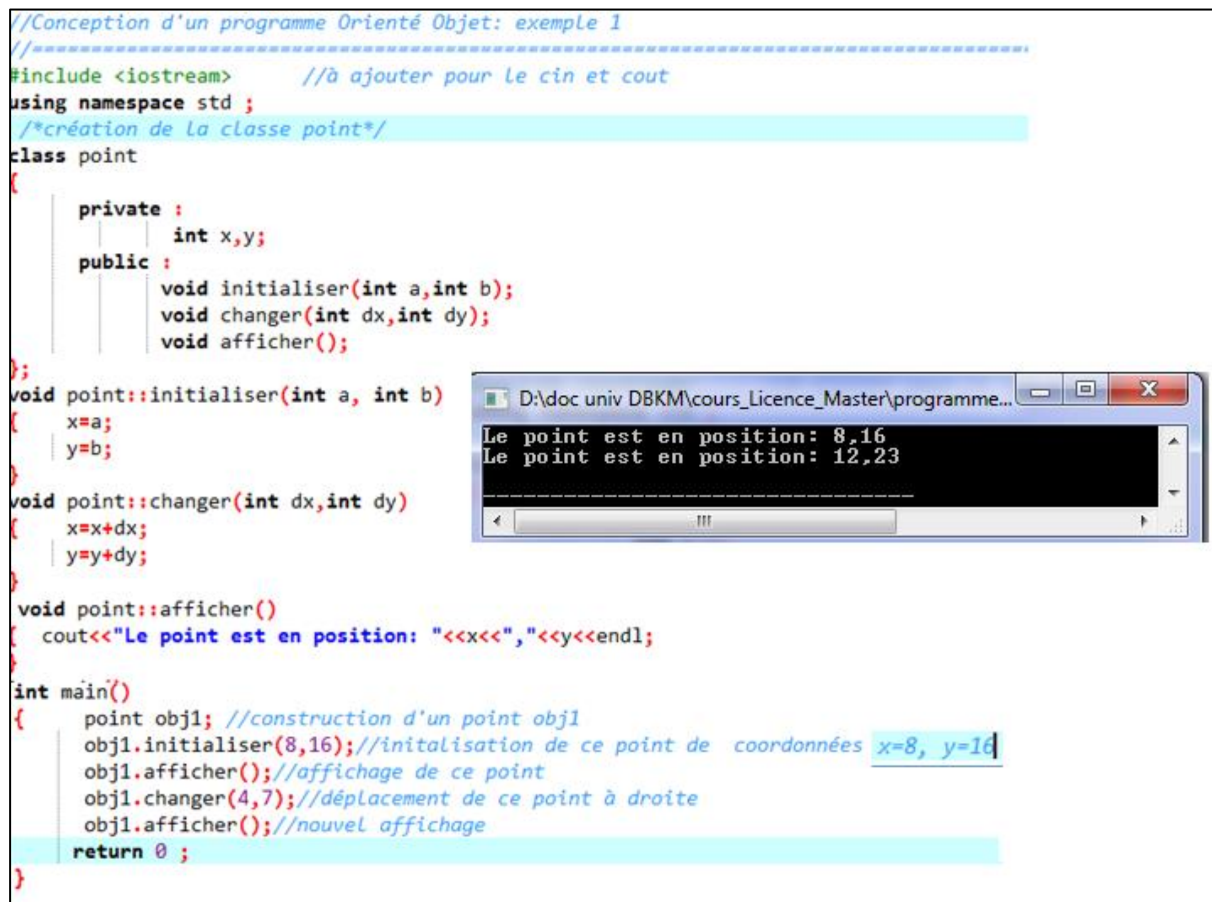
```
D:\doc univ DBKM\cours_Licence_Master\programme_AUTO_ELN_EL...
constrcteur=1
constrcteur=2
Le cercle1 est en position: 10,2
de rayon: 5
de surface: 78.5398
de circonference: 31.4159
Le cercle2 est en position: 20,4
de rayon: 10
de surface: 314.159
de circonference: 62.8319
Nombre d'objets circle restants : 1
Nombre d'objets circle restants : 0
```

Exercice 2

Nous réalisons une classe **Point** permettant de manipuler un point dans un plan. Cette classe comporte deux membres données privées : x et y et trois fonctions membres (ou méthodes) publiques : **initialiser, deplacer, afficher.**

D'abord, on déclare la classe au début du programme après le mot clé class, puis on définit le contenu des méthodes membres.

obj1 est un objet de classe point.



```
//Conception d'un programme Orienté Objet: exemple 1
//====================================================================
#include <iostream>      //à ajouter pour le cin et cout
using namespace std ;
 /*création de la classe point*/
class point
{
    private :
        int x,y;
    public :
        void initialiser(int a,int b);
        void changer(int dx,int dy);
        void afficher();
};
void point::initialiser(int a, int b)
{    x=a;
     y=b;
}
void point::changer(int dx,int dy)
{    x=x+dx;
     y=y+dy;
}
 void point::afficher()
{  cout<<"Le point est en position: "<<x<<","<<y<<endl;
}
int main()
{     point obj1; //construction d'un point obj1
      obj1.initialiser(8,16);//initalisation de ce point de  coordonnées x=8, y=16
      obj1.afficher();//affichage de ce point
      obj1.changer(4,7);//déplacement de ce point à droite
      obj1.afficher();//nouvel affichage
     return 0 ;
}
```

D:\doc univ DBKM\cours_Licence_Master\programme...

```
Le point est en position: 8,16
Le point est en position: 12,23
```

Exercice 3

En utilisant le code fourni ci-dessus sur la conception de la classe point, pour instancier plusieurs objets de type point :

1. Créez un objet obj2 de type point avec les valeurs suivantes : (3,5).
2. Faites un déplacement de obj2 à droite de (10,12).
3. Ajoutez une méthode changerG qui déplace le point à gauche.
4. Faites un déplacement de obj2 à gauche de (10,12).
5. Créez un objet obj3 de type point avec les valeurs suivantes : (17,17).
6. Faites un déplacement de obj3 à gauche de (7,7) et à droite de (17,17).

```cpp
//==================================================================
#include <iostream>
using namespace std ;
 //création de la classe point
class point
{
private :
    int x,y;
public :
      void initialiser(int a,int b);
      void changerD(int dx,int dy);
      void changerG(int dx,int dy);
      void afficher(int n);
};
void point::initialiser(int a, int b)
{
     x=a;
     y=b;
}
void point::changerD(int dx,int dy)
{
     x=x+dx;
     y=y+dy;
}
void point::changerG(int dx,int dy)
{
     x=x-dx;
     y=y-dy;
}
void point::afficher(int n)
{
     cout<<"Le point "<<n<<" est en position: "<<x<<","<<y<<endl;
}
int main()
{
     point obj1,obj2,obj3;        //déclaration d'un point obj1
     obj1.initialiser(8,16); //initalisation de ce point de coordonnées x=8, y=16
     obj1.afficher(1);

     obj2.initialiser(3,5);
     obj2.afficher(2);          //affichage du point obj2

     obj3.initialiser(17,17);
     obj3.afficher(3);          //affichage du point obj3

     obj1.changerD(2,3); //déplacement de obj1 à droite
     obj1.afficher(1);

     obj2.changerD(10,12);
     obj2.afficher(2);  //nouvel affichage de obj2

     obj2.changerG(10,12);
     obj2.afficher(2);

     obj3.changerG(7,7);
     obj3.afficher(3);

     obj3.changerD(17,17);
     obj3.afficher(3);
     return 0;
}
```

```
D:\doc univ DBKM\cours_Licence_M...
Le point 1 est en position: 8,16
Le point 2 est en position: 3,5
Le point 3 est en position: 17,17
Le point 1 est en position: 10,19
Le point 2 est en position: 13,17
Le point 2 est en position: 3,5
Le point 3 est en position: 10,10
Le point 3 est en position: 27,27
```

Exemple 4

Définir une classe rectangle représentant une abstraction (informatique) de ce qu'est un rectangle : une largeur, une longueur, sa surface et son circonférence. Cette classe contient un emplacement x et y une **longueur** et une **largeur** comme variables, les fonctions **initialiser, deplacer, surface, circonference** et **afficher** sont leurs méthodes.

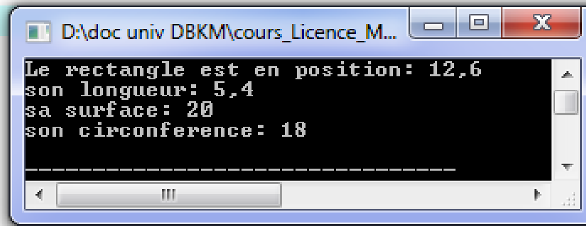
- La fonction **initialiser** permet d'initialiser la largeur et longueur x, y.
- La fonction **deplacer** permet de déplace le rectangle par un décalage à droite de dx et dy.
- La fonction surface permet de renvoyer la valeur la surface du rectangle.
- La fonction **circonference** permet de renvoyer la valeur de la circonférence du rectangle.
- La fonction **afficher,** permet d'affiche la position, la longueur, la largeur, la surface ainsi que le périmètre du rectangle.

D'abord on doit définir une classe rectangle permettant de manipuler les fonctions définies ci-dessus. Utilisez cette classe pour instancier un objet rec1 :

a) Crée un objet rec1 de type rectangle.

b) Initialise le rec1 par les valeurs (10,2,5,4).

c) Fait un déplacement de rec1 de (2,4).
d) Calcule la surface et le périmètre de rec1.
e) Affiche les résultats.




```
//==================================================================
#include <iostream>
using namespace std;
 class rectangle
{
    private :
        int x,y;
        float lon,lar;
    public :
        void initialiser(int a,int b, int c, int d);
        void deplacer(int dx, int dy);
        float surface();
        float circonference();
        void afficher();
};
void rectangle::initialiser(int a, int b, int c, int d)
{
    x=a;
    y=b;
    lon=c;
    lar=d;
}
void rectangle::deplacer(int dx,int dy)
{
    x=x+dx;
    y=y+dy;
}
float rectangle::surface()
{
    return lon*lar;
}
float rectangle::circonference()
{
    return 2*(lon+lar);
}
void rectangle::afficher()
{
    cout<<"Le rectangle est en position: "<<x<<","<<y<<"\n"
    <<"son longueur: "<<lon<<","<<lar<<"\n"
    <<"sa surface: "<<surface()<<"\n"
    <<"son circonference: "<<circonference()<<endl;
}
    <<"sa surface: "<<surface()<<"\n"
    <<"son circonference: "<<circonference()<<endl;
}
int main()
{
    rectangle rec1;
    rec1.initialiser(10,2,5,4);
    rec1.deplacer(2,4);
    rec1.surface();
    rec1.circonference();
    rec1.afficher();
       return 0;
}
```