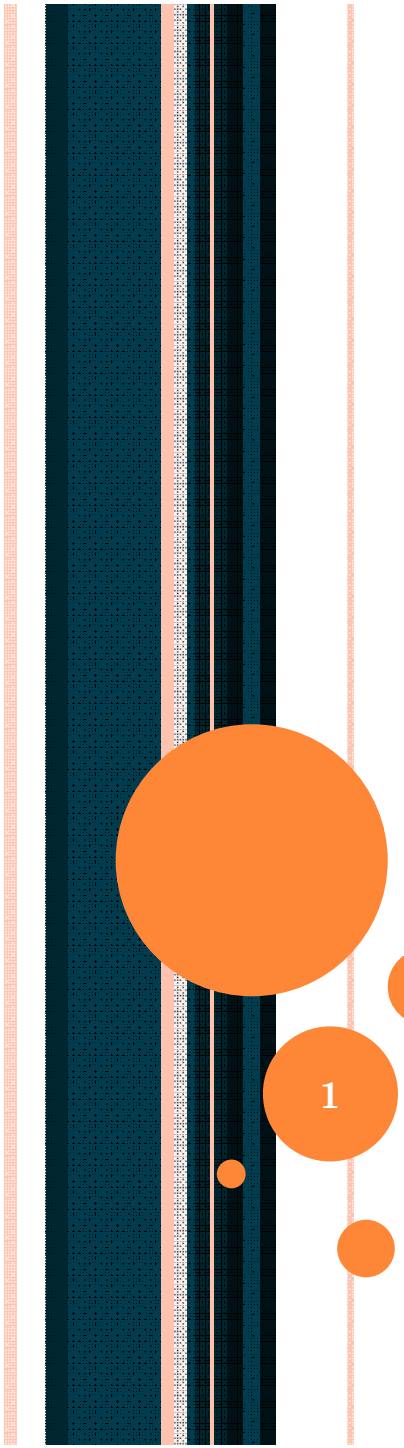


LANGAGE C++

Programmation Orientée Objet

CHAPITRE1: GÉNÉRALITÉS SUR LE LANGAGE C++

- **Voici un exemple de programme en langage C++.**

```
#include <iostream>
#include <cmath>
using namespace std ;
main(){ int i ; float x ; float racx ; const int NFOIS = 5 ;
cout << "Bonjour\n" ;
cout << "Je vais vous calculer " << NFOIS << " racines carrees\n" ;
for (i=0 ; i<NFOIS ; i++)
{ cout << "Donnez un nombre : " ;   cin >> x ;
if (x < 0.0)  cout << "Le nombre " << x << "ne possède pas de racine
                carree\n " ;
else  { racx = sqrt (x) ;
        cout << "Le nombre " << x << " a pour racine carree : " << racx
        << "\n" ; } }
cout << "Travail termine - au revoir " ; }
```

Exemple d'exécution

- Bonjour
- Je vais vous calculer 5 racines carrees
- Donnez un nombre : 8
- Le nombre 8 a pour racine carree : 2.82843
- Donnez un nombre : 4
- Le nombre 4 a pour racine carree : 2
- Donnez un nombre : 0.25
- Le nombre 0.25 a pour racine carree : 0.5
- Donnez un nombre : 3.4
- Le nombre 3.4 a pour racine carree : 1.84391
- Donnez un nombre : 2
- Le nombre 2 a pour racine carree : 1.41421
- Travail termine - au revoir

STRUCTURE D'UN PROGRAMME EN LANGAGE C++

- ***main()*** c'est comme celle du langage C.
- ***cout*** est un « flot de sortie »
- et que **<<** est un opérateur permettant d'envoyer de l'information sur un flot de sortie.
- **cin** est un « flot d'entrée » associé au clavier
- et que **>>** est un opérateur permettant d'«extraire» (de lire) de l'information à partir d'un flot d'entrée.
- ***iostream*** contient des déclarations relatives aux flots donc, en particulier, à **cin** et ***cout***, ainsi qu'aux opérateurs **<<** et **>>**
- ***cmath*** contient des déclarations relatives aux fonctions mathématiques (héritées de C), donc en particulièr à ***sqrt***.

L'instruction *using*

- Pour l'instant, retenez que les symboles déclarés dans le fichier ***iostream*** appartiennent à l'espace de noms ***std***.
- L'instruction ***using*** sert précisément à indiquer que l'on se place « dans cet espace de noms **std** »

LE TYPE BOOL

- Ce type est tout naturellement formé de deux valeurs notées ***true*** et ***false***. Il peut intervenir dans des constructions telles que :

- bool ok = false ;

.....

if (.....) ok = true ;

.....

if (ok) .....

CHAPITRE 2 : LES ENTRÉES-SORTIES CONVERSATIONNELLES DE C++

AFFICHAGE À L'ÉCRAN

- **Exemple 1 :** Considérons ces instructions :

```
int n = 25 ;
cout << "valeur : " ;
cout << n ;
```

- Elles affichent le résultat suivant : ***valeur : 25***
- Le rôle de l'opérateur << est manifestement différent dans les deux cas :
  - dans le premier, il a transmis les caractères de la chaîne,
  - dans le second, il a procédé à un «formatage » pour convertir une valeur entière en une suite de caractères.

AFFICHAGE À L'ÉCRAN

- **Exemple 2 :** Les deux instructions :

cout << "valeur :" ;

cout << n ;

peuvent se condenser en une seule :

cout << "valeur :" << n ;

- Celle-ci peut s'interpréter comme ceci :
  - dans un 1er temps, le flot **cout** reçoit la chaîne "***valeur*** :" ;
  - dans un 2ème temps, le flot ***cout << "valeur :",*** c'est-à-dire le flot **cout** augmenté de "***valeur*** :", reçoit la valeur de ***n***.

AFFICHAGE À L'ÉCRAN

- Rappelons que les déclarations nécessaires à l'utilisation des opérateurs **<<** et **>>** figurent dans un fichier en-tête de nom ***<iostream>***
- et que les symboles correspondants sont définis dans l'espace de noms ***std***.
- On vous a indiqué qu'elle vous obligeait à introduire une instruction ***using***, et donc à commencer tous vos programmes par :
- ***#include <iostream>***
- ***using namespace std ;*** /*on utilisera les symboles définis dans */

/* l'espace de noms standard s'appelant **std** */

LECTURE AU CLAVIER

- **>>** était un opérateur permettant de lire de l'information sur le flot ***cin*** correspondant au clavier.
- **Exemple :**

int n ; char c ;

- ***cin >> n ;*** // lit une suite de caractères représentant un entier, la convertit en ***int*** et range le résultat dans ***n***
- ***cin >> c ;*** // lit un caractère et le range dans ***c***
- il possède une associativité de gauche à droite.
- ***cin >> n >> p ;*** sera équivalent à : ***(cin >> n) >> p ;***

CHAPITRE 3 : LES FONCTIONS

1 Transmission des arguments par valeur

- **Exemple 1 :**

```
#include <iostream>
using namespace std ;
main()   { void echange (int a, int b) ;  int n=10, p=20 ;
cout << "avant appel : " << n << " " << p << "\n" ;
echange (n, p) ;
cout << "apres appel : " << n << " " << p << "\n" ; }
void echange (int a, int b) { int c ;
cout << "début echange : "<< a << " " << b << "\n" ;
c = a ;  a = b ;  b = c ;
cout << "fin echange : " << a << " " << b << "\n" ; }
```

1 Transmission des arguments par valeur

- **Résultat** :

avant appel : 10 20

debut echange : 10 20

fin echange : 20 10

apres appel : 10 20

2 TRANSMISSION PAR RÉFÉRENCE

- C++ dispose de la notion de **référence**, laquelle correspond à celle **d'adresse** : considérer la référence d'une variable revient à considérer son adresse, et non plus sa valeur.
- **Exemple 2** :

```
#include <iostream>
using namespace std ;
main() { void echange (int &, int &) ;  int n=10, p=20 ;
cout << "avant appel : " << n << " " << p << "\n" ;
echange (n, p) ; // attention, ici pas de &n, &p
cout << "apres appel : " << n << " " << p << "\n" ; }
void echange (int & a, int & b) { int c ;
cout << "debut echange : " << a << " " << b << "\n" ;
c = a ; a = b ; b = c ;
cout << "fin echange : " << a << " " << b << "\n" ; }
```

2 Transmission par référence (suite)

- **Résultat** :

avant appel : 10 20

début echange : 10 20

fin echange : 20 10

après appel : 20 10

3 SURDÉFINITION DE FONCTIONS

- D'une manière générale, on parle de « **surdéfinition** » lorsqu'un même symbole possède plusieurs significations différentes.
- Le choix de l'une des significations se faisant en fonction du contexte.
- Pour pouvoir employer plusieurs fonctions de même nom, il faut bien sûr un critère (autre que le nom) permettant de choisir la bonne fonction.
- En C++, ce choix est basé sur le type des arguments.

EXEMPLE 3

```
#include <iostream>
using namespace std ;
void sosie (int) ;    // les prototypes
void sosie (double) ;
main() // le programme de test
{ int n=5 ; double x=2.5 ;
sosie (n) ;
sosie (x) ; }
void sosie (int a)    // la première fonction
{ cout << "sosie numero I a = " << a << "\n" ; }
void sosie (double a) // la deuxième fonction
{ cout << "sosie numero II a = " << a << "\n" ; }
```

RÉSULTAT

- sosie numero I **a = 5**
- sosie numero II **a = 2.5**

- Vous constatez que le compilateur a bien mis en place l'appel de la « bonne fonction » **sosie**, au vu de la liste d'arguments (ici réduite à un seul).

Exemple de Surdéfinition de fonctions

- Soient les déclarations (C++) suivantes :

```
int fct (int) ;                 // fonction I
int fct (float) ;               // fonction II
void fct (int, float) ;         // fonction III
void fct (float, int) ;         // fonction IV
int n, p ;    float x, y ;    char c ;    double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

- **a.** fct (n) ;
- **b.** fct (x) ;
- **c.** fct (n, x) ;
- **d.** fct (x, n) ;
- **e.** fct (c) ;
- **f.** fct (n, p) ;
- **g.** fct (n, c) ;
- **h.** fct (n, z) ;
- **i.** fct (z, z) ;

SOLUTION

- Les cas **a**, **b**, **c** et **d** ne posent aucun problème. Il y a respectivement appel des fonctions I, II, III et IV, sans qu'aucune conversion d'argument ne soit nécessaire.
- **e.** Appel de la fonction I, après conversion de la valeur de c en int.
- **f.** Appel, compte tenu de son **ambiguïté** ; deux possibilités existent **incorrect** en effet : conserver n, convertir p en float et appeler la fonction III ou, au contraire, convertir n en float, conserver p et appeler la fonction IV.
- **g.** Appel de la fonction III, après conversion de c en float.
- **h.** Appel de la fonction III, après conversion (dégradante) de z en float.
- **i.** Appel **incorrect**, compte tenu de son **ambiguïté** ; deux possibilités existent en effet : convertir le premier argument en float et le second en int et appeler la fonction III ou, au contraire, convertir le premier argument en int et le second en float et appeler la fonction IV. Notez que, dans les deux cas, il s'agit de conversions dégradantes.

Notion de Référence Et Initialisation

- Considérez, par exemple, ces instructions :

int n ; int &p = n ;

- La seconde signifie que **p** est une référence à la variable **n**. Ainsi, dans la suite, **n** et **p** désigneront le même emplacement mémoire. Par exemple, avec :

  n = 3 ; cout << p ; nous obtiendrons la valeur **3**.

- La déclaration : **int &p = n ;** est en fait une déclaration de référence (ici **p**) accompagnée d'une initialisation (à la référence de **n**).
- D'une façon générale, il n'est pas possible de déclarer une référence sans l'initialiser, comme dans :
- **int &p ; // incorrect, car pas d'initialisation**

CHAPITRE 4 : LA GESTION DYNAMIQUE : LES OPÉRATEURS *new* ET *delete*

- **L'opérateur *new***
- **Exemple 1 :**

  int *ad ;

  ad = new int ; cette instruction permet d'allouer l'espace mémoire nécessaire pour un élément de type **int** et d'affecter à **ad** l'adresse correspondante.
- **Ou bien : int *ad = new int ;**

- **Exemple 2 :**

  char *adc ;

  adc = new char[100] ; cette instruction alloue l'emplacement nécessaire pour un tableau de **100 caractères** et place l'adresse (de début) dans **adc**.

SYNTAXE DE *new*

- ***new*** s'utilise ainsi : ***new type*** où **type** représente un type absolument quelconque. Il fournit comme résultat un pointeur (de type **type***) sur l'emplacement correspondant, lorsque l'allocation a réussi.

- L'opérateur ***new*** accepte également une syntaxe de la forme : ***new type [n]*** Où **n** désigne une expression entière quelconque. Cette instruction alloue alors l'emplacement nécessaire pour **n éléments** du type indiqué ; si l'opération a réussi, elle fournit en résultat un pointeur (toujours de type **type ***) sur le premier élément de ce tableau.

L'OPÉRATEUR *delete*

- Lorsque l'on souhaite libérer un emplacement alloué préalablement par ***new***, on utilise l'opérateur ***delete***.
- **Exemple :**
- ***delete ad ;*** pour l'emplacement alloué par: ***ad = new int ;***
- ***delete adc ;*** pour l'emplacement alloué par :
  adc = new char [100] ;
- ***delete adresse*** (**adresse** étant une expression devant avoir comme valeur un pointeur sur un emplacement alloué par **new**)
- Notez bien que le comportement du programme n'est absolument pas défini lorsque :
  - vous libérez par ***delete*** un emplacement déjà libéré ;
  - vous fournissez à ***delete*** une « mauvaise adresse » ou un pointeur obtenu autrement que par ***new***.
- Autre syntaxe de ***delete***, de la forme ***delete [] adresse***, qui n'intervient que dans le cas de tableaux d'objets

EXEMPLE 1 : new ET delete

- Écrire plus simplement en C++ les instructions suivantes, en utilisant les opérateurs new et delete :

```
int * adi ;
double * add ;
.....
adi = malloc (sizeof (int) ) ;
add = malloc (sizeof (double) * 100 ) ;
```

SOLUTION 1 : new ET delete

```
int * adi ;
double * add ;
.....
adi = new int ;
add = new double [100] ;
```

- Ou bien

```
int * adi = new int ;
double * add = new double [100] ;
```

EXEMPLE 2 : new ET delete

- Écrire plus simplement en C++, en utilisant les spécificités de ce langage, les instructions C suivantes :

```
double * adtab ;
int nval ;
.....
printf ("combien de valeurs ? ") ;
scanf ("%d", &nval) ;
adtab = malloc (sizeof (double) * nval) ;
```

SOLUTION 2 : new ET delete

```
double * adtab ;
int nval ;
.....
cout << "combien de valeurs ? " ;
cin >> nval ;
adtab = new double [nval] ;
```

SPÉCIFICATION *inline*

- ***Rappel sur les macros en C***
- En C, on peut définir une macro avec le mot-clé define.

  Exemple : #define carre(A) A*A
- Cela permet à priori d'utiliser **carre** comme une fonction normale :

```
int a = 2 ;
int b = carre(a) ;
cout << "  a = " << a << " b= " << b << endl ;
```

- Le résultat sera correct dans ce cas simple :

  a = 2 b = 4
- Quand il voit une macro, le C remplace partout dans le code les expressions **carre(x)** par **x*x**

SPÉCIFICATION *inline*

- L'avantage d'une macro par rapport à une fonction est la rapidité d'exécution. En effet, le temps à recopier les valeurs des paramètres disparait.
- La contrepartie est un espace mémoire du programme plus grand.
- Plus embêtant sont les effets de bord des macros. Si l'on programme : **b = carre(a++) ;**
- On attendrait le résultat suivant : **a = 3 b = 4**

  car **b = carre(a++) ; <==> b = carre(a) ; a++ ;**
- En réalité, ce n'est pas le cas car, à la compilation, le C remplace malheureusement **carre(a++)** par **a++ * a++**. Et à l'exécution, le programme donne le résultat suivant : **a=4 b=4**

SPÉCIFICATION *inline*

- ***Les fonctions inline***
- Le C++ palie à cet inconvénient en permettant de définir des fonctions, dites en ligne, avec le mot-clé ***inline***.

```
inline  int  carre(int x)  { return x*x ; }
```

- Une fonction définie avec le mot-clé ***inline*** aura le même avantage qu'une macro en C, à savoir un gain de temps d'exécution et le même inconvénient, à savoir une plus grande place mémoire occupée par le programme.
- L'avantage des fonctions en ligne est la disparition des effets de bord.

SPÉCIFICATION *inline*

- **Exercice :** Transformer le programme suivant pour que la fonction **fct** devienne une fonction en ligne.

```
#include <iostream>
using namespace std ;
main() { int fct (char, int) ; // déclaration (prototype) de fct
int n = 150, p ;    char c = 's' ;   p = fct ( c , n) ;
cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;  }

int fct (char c, int n)                    // définition de fct
{ int res ;
  if (c == 'a')   res = n + c ;
  else     if (c == 's')   res = n - c ;
           else   res = n * c ;
return res ;  }
```

SPÉCIFICATION *inline*

- **Solution** : Nous devons donc d'abord déclarer (et définir en même temps) la fonction **fct** comme une fonction en ligne. Le programme **main** s'écrit de la même manière, si ce n'est que la déclaration de **fct** n'y est plus nécessaire puisqu'elle apparaît auparavant (il reste permis de la déclarer, à condition de ne pas utiliser le qualificatif **inline**).

```
#include <iostream>
using namespace std ;
inline int fct (char c, int n) { int res ;
                                 if (c == 'a') res = n + c ;
                                 else   if (c == 's') res = n - c ;
                                        else res = n * c ;
                                 return res ;  }
main () { int n = 150,  p ;  char c = 's' ;
          p = fct (c, n) ;
          cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ; }
```

CHAPITRE 5 : CLASSES ET OBJETS

Les structures généralisées

Soit la structure ***struct point { int x ;***

int y ; }

- C++ permet de lui associer des méthodes (fonctions membres).
- Supposons que nous souhaitions introduire trois fonctions :
  - ***initialise*** pour attribuer des valeurs aux « coordonnées » d'un point ;
  - ***deplace*** pour modifier les coordonnées d'un point ;
  - ***affiche*** pour afficher un point : ici, nous nous contenterons, par souci de simplicité, d'afficher les coordonnées du point.

FONCTIONS MEMBRES D'UNE STRUCTURE

- Voici comment nous pourrions déclarer notre structure point :

```
struct point
{ /* déclaration "classique" des données */
int x ;
int y ;
/* déclaration des fonctions membres (méthodes) */
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```

FONCTIONS MEMBRES D'UNE STRUCTURE

- ***void point::initialise (int abs, int ord)***
  { x = abs ; y = ord ; }
- Dans l'en-tête, le nom de la fonction est : ***point::initialise***
- Le symbole **::** correspond à ce que l'on nomme l'opérateur de «résolution de portée», lequel sert à modifier la portée d'un identificateur.
- Ici, il signifie que l'identificateur ***initialise*** concerné est celui défini dans ***point***.
- En l'absence de ce « préfixe » (***point::***), nous définirions effectivement une fonction nommée ***initialise***, mais celle-ci ne serait plus associée à ***point*** ; il s'agirait d'une fonction « ordinaire » nommée ***initialise***, et non plus de la fonction membre ***initialise*** de ***la structure point***.

FONCTIONS MEMBRES D'UNE STRUCTURE

- Dans le corps de la fonction ***initialise***, On trouve une affectation : ***x = abs*** ;
  - Le symbole ***abs*** désigne la valeur reçue en premier argument.
  - Mais ***x***, quant à lui, n'est ni un argument ni une variable locale.
- En fait, ***x*** désigne **le membre *x*** correspondant au type ***point*** (cette association étant réalisée par le ***point::*** de l'en-tête).

EXEMPLE RÉCAPITULATIF

Exemple 1 :

```
#include <iostream>
using namespace std ;
/* ----------- Déclaration du type point ------------ */
struct point  { /* déclaration "classique" des données */
                    int x ;    int y ;
/* déclaration des fonctions membres (méthodes) */
void initialise (int, int) ;   void deplace (int, int) ;  void affiche () ;  };

/* ---- Définition des fonctions membres du type point --- */
void point::initialise (int abs, int ord)   { x = abs ;    y = ord ;  }
void point::deplace (int dx, int dy)   { x += dx ;    y += dy ; }
void point::affiche () { cout << "Je suis en " << x << " " << y << "\n" ; }
main() { point a, b ;
a.initialise (5, 2) ; a.affiche() ;
a.deplace (-2, 4)  ; a.affiche() ;
b.initialise (1,-1) ; b.affiche() ;
}
```

Résultat de l'exemple

Je suis en 5 2

Je suis en 3 6

Je suis en 1 -1

NOTION DE CLASSE

- En C++, une classe est une structure dans laquelle seulement certains membres et/ou fonctions membres seront «**publics**», c'est-à-dire accessibles « de l'extérieur », les autres membres étant dits « **privés** ».

- La déclaration d'une classe est voisine de celle d'une structure. En effet, il suffit :
  - de remplacer le mot clé **struct** par le mot clé **class** ;
  - de préciser quels sont les membres publics (fonctions ou données) et les membres privés en utilisant les mots clés **public** et **private**.

NOTION DE CLASSE

- **Exemple :**

```
/* ---------- Déclaration de la classe point ------------ */
class point
{                    /* déclaration des membres privés */
private :      /* facultatif */
int x ;
int y ;
                     /* déclaration des membres publics */
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```

NOTION DE CLASSE

Exemple 2 :

```
#include <iostream>
using namespace std ;
/* ---------- Déclaration de la classe point ------------ */
class point { /* déclaration des membres privés */
private :   int x ;   int y ;
/* déclaration des membres publics */
public :  void initialise (int, int) ;  void deplace (int, int) ;
              void affiche () ; } ;
/* - Définition des fonctions membres de la classe point - */
void point::initialise (int abs, int ord)  { x = abs ; y = ord ;}
void point::deplace (int dx, int dy)  { x = x + dx ; y = y + dy ;}
void point::affiche ()  { cout << "Je suis en " << x << " " << y << "\n" ;}
/* -------- Utilisation de la classe point -------- */
main()  {  point a, b ;
a.initialise (5, 2) ; a.affiche () ;
a.deplace (-2, 4) ; a.affiche () ;
b.initialise (1,-1) ; b.affiche () ;  }
```

NOTION DE CLASSE

- Dans le jargon de la P.O.O., on dit que **a** et **b** sont des **instances** de **la *classe point***,
- ou encore que ce sont des **objets** de **type *point*** ; c'est généralement ce dernier terme que nous utiliserons.
- Il existe un troisième mot, ***protected*** (**protégé**), qui s'utilise de la même manière que les deux autres ; il sert à définir un statut intermédiaire entre **public** et **privé**, lequel n'intervient que dans le cas de **classes dérivées**.
- On peut définir des classes anonymes, comme on pouvait définir des structures anonymes.

AFFECTATION D'OBJETS

- Soit ***struct point { int x ; int y ; } ;***
- ***point a, b ;***
- Vous pouvez écrire : ***b = a ;*** Cette instruction recopie l'ensemble des valeurs des champs de ***a*** dans ceux de ***b***. Elle joue le même rôle que : ***b.x = a.x ; b.y = a.y ;***
- Cette possibilité s'étend aussi aux objets de même type. Elle correspond tout naturellement à une recopie des valeurs des membres données, que ceux-ci soient publics ou non.
- ***class point { int x ;***

  public : int y ; .... } ;
- ***point a, b ;***
- L'instruction : ***b = a ;*** provoquera la recopie des valeurs des membres ***x*** et ***y*** de ***a*** dans les membres correspondants de ***b***.

AFFECTATION D'OBJETS

- Contrairement à ce qui a été dit pour les structures, il n'est plus possible ici de remplacer cette instruction par : ***b.x = a.x ; b.y = a.y ;***
- En effet,
  - si **la deuxième affectation est légale**, puisque ici ***y*** est **public**,
  - **la première ne l'est pas**, car ***x*** est **privé**.

- *L'affectation **a = b** est toujours légale, quel que soit le statut (**public** ou **privé**) des membres données. On peut considérer qu'elle ne viole pas le principe d'encapsulation, dans la mesure où les données **privées** de **b** (les copies de celles de **a**, après affectation) restent toujours inaccessibles de manière directe.*

.

NOTIONS DE CONSTRUCTEUR ET DE DESTRUCTEUR

- Le **constructeur** est une fonction membre (définie comme les autres fonctions membres) qui sera appelée automatiquement à chaque création d'un objet.
- Le **destructeur**, est une fonction membre appelée automatiquement au moment de la destruction de l'objet.
- Par convention, le **constructeur** se reconnaît à ce qu'il porte le même nom que la classe.
- Quant au **destructeur**, il porte le même nom que la classe, précédé d'un tilde (~).

EXEMPLE DE CLASSE COMPORTANT UN CONSTRUCTEUR

- Considérons la classe ***point*** précédente et transformons simplement notre fonction membre **initialise** en un constructeur en la renommant **point** (dans sa déclaration et dans sa définition).

- La déclaration de notre nouvelle classe ***point*** se présente ainsi :

```
class point
{       /* déclaration des membres privés */
int x ;
int y ;
public :  /* déclaration des membres publics */
point (int, int) ;   // constructeur
void deplace (int, int) ;
void affiche () ;
} ;
```

- À partir du moment où une classe possède un constructeur, **il n'est plus possible de créer un objet sans fournir les arguments requis par son constructeur** (sauf si ce dernier ne possède aucun argument !).

EXEMPLE D'UTILISATION DE CONSTRUCTEUR

Exemple 3 :

```
#include <iostream>
using namespace std ;
/* ---------- Déclaration de la classe point ------------ */
class point {       /* déclaration des membres privés */
int x ;  int y ;
        /* déclaration des membres publics */
public :    point (int, int) ;   //    constructeur
void deplace (int, int) ;
void affiche () ;  } ;

/* -- Définition des fonctions membre de la classe point -- */
point::point (int abs, int ord)  { x = abs ; y = ord ;}
void point::deplace (int dx, int dy)  { x = x + dx ; y = y + dy ;}
void point::affiche ()  { cout << "Je suis en " << x << " " << y << "\n" ;}

/* -------- Utilisation de la classe point -------- */
main()  { point a(5,2) ;  a.affiche () ;  a.deplace (-2, 4) ; a.affiche () ;
point b(1,-1) ;  b.affiche () ;  }
```

CONSTRUCTION ET DESTRUCTION DES OBJETS

Exemple 4 :

```
#include <iostream>
using namespace std ;
class test { public : int num ;
                      test (int) ;   // déclaration constructeur
                     ~test () ;      // déclaration destructeur  } ;
test::test (int n)  // définition constructeur
{ num = n ;
cout << "++ Appel constructeur - num = " << num << "\n" ;  }
test::~test ()      // définition destructeur
{ cout << "- - Appel destructeur - num = " << num << "\n" ;  }
main()  { void fct (int) ;
test a(1) ;
for (int i=1 ; i<=2 ; i++) fct(i) ;  }
void fct (int p)
{ test x(2*p) ;   // notez l'expression (non constante) : 2*p }
```

Résultat du Programme

++ Appel constructeur - num = 1

++ Appel constructeur - num = 2

-- Appel destructeur - num = 2

++ Appel constructeur - num = 4

-- Appel destructeur - num = 4

-- Appel destructeur - num = 1

- L'objet **a** est détruit à la sortie du programme **main()**
- et les objets **x** sont détruits à la sortie de la fonction **fct**.

EXEMPLE D'APPLICATION N° 1

- Un programme exploitant une classe nommée ***hasard***, dans laquelle le constructeur fabrique dix valeurs entières aléatoires qu'il range dans le membre donnée ***val*** (ces valeurs sont comprises entre **zéro** et **la valeur qui lui est fournie en argument**)

EXEMPLE D'APPLICATION N° 1 (SUITE)

```
#include <iostream>
#include <cstdlib>     // pour la fonction rand
using namespace std ;
class hasard   { int val[10] ;
                 public :    hasard (int) ;
                             void affiche () ;   } ;
hasard::hasard (int max)
      // constructeur : il tire 10 valeurs au hasard
      // rappel : rand fournit un entier entre 0 et RAND_MAX
{ int i ;   for (i=0 ; i<10 ; i++) val[i]=double(rand())/RAND_MAX * max ;  }
void hasard::affiche () // pour afficher les 10 valeurs
{ int i ;
 for (i=0 ; i<10 ; i++) cout << val[i] << " " ;
 cout << "\n" ;  }
main()
{ hasard suite1 (5) ;  suite1.affiche () ;
  hasard suite2 (12) ;  suite2.affiche () ;  }
```

Exemple d'Application N° 1 (suite)

- **Résultat** :

0 2 0 4 2 2 1 4 4 3

2 10 8 6 3 0 1 4 1 1

- Par ailleurs, à partir du moment où un emplacement a été **alloué** **dynamiquement**, il faut se soucier de sa **libération** lorsqu'il sera devenu inutile.
- Là encore, il paraît tout naturel de confier ce travail au **destructeur** de la classe.

EXEMPLE D'APPLICATION N° 2

EXEMPLE DE CLASSE DONT LE CONSTRUCTEUR EFFECTUE UNE ALLOCATION DYNAMIQUE DE MÉMOIRE

```
#include <iostream>
#include <cstdlib>    // pour la fonction rand
using namespace std ;
class hasard   { int nbval ;        // nombre de valeurs
                 int * val ;        // pointeur sur les valeurs
                public :  hasard (int, int) ;   // constructeur
                          ~hasard () ;          // destructeur
                          void affiche () ; } ;
hasard::hasard (int nb, int max)   { int i ;     val = new int [nbval = nb] ;
for (i=0 ; i<nb ; i++) val[i] = double (rand()) / RAND_MAX * max ;   }
hasard::~hasard ()  { delete val ;  }
void hasard::affiche ()  // pour afficher les nbavl valeurs
{ int i ;
for (i=0 ; i<nbval ; i++) cout << val[i] << " " ;  cout << "\n" ;  }
main()  {  hasard suite1 (10, 5) ;  // 10 valeurs entre 0 et 5
           suite1.affiche () ;
           hasard suite2 (6, 12) ;  // 6 valeurs entre 0 et 12
           suite2.affiche () ;  }
```

EXEMPLE D'APPLICATION N° 2 (SUITE)

- **Résultat**

0 2 0 4 2 2 1 4 4 3

2 10 8 6 3 0

- Dans le constructeur, l'instruction :

  val = new [nbval = nb] ; joue le même rôle que :

  nbval = nb ;

  val = new [nbval] ;

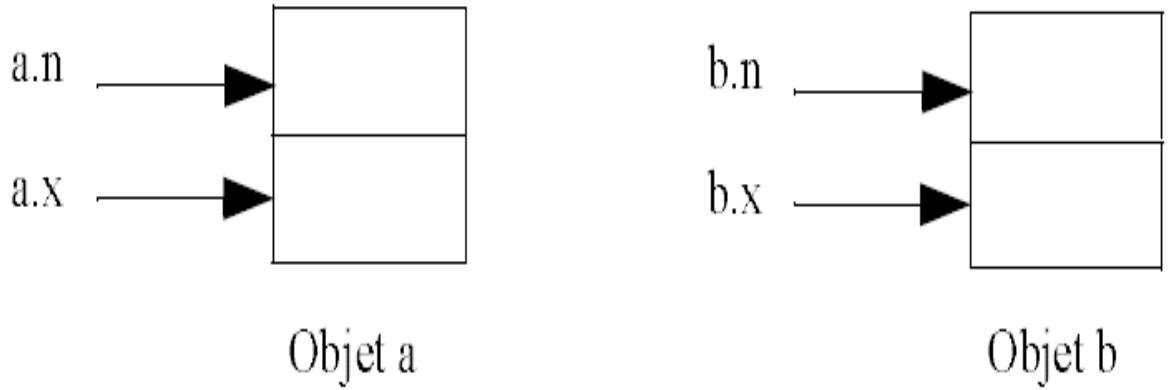
Quelques règles concernant Constructeur et Destructeur

- Un constructeur peut comporter un nombre quelconque d'arguments, éventuellement aucun.
- Par définition, un constructeur ne renvoie pas de valeur ; aucun type ne peut figurer devant son nom (la présence de **void** est une erreur).
- Par définition, un destructeur ne peut pas disposer d'arguments et ne renvoie pas de valeur. Là encore, aucun type ne peut figurer devant son nom (et la présence de **void** est une erreur).
- En théorie, constructeurs et destructeurs peuvent être publics ou privés. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre publics.

LES MEMBRES DONNÉES STATIQUES

- lorsque dans un même programme on crée différents objets d'une même classe, chaque objet possède ses propres membres données.
- Par exemple, si nous avons défini une classe ***exple1*** par :

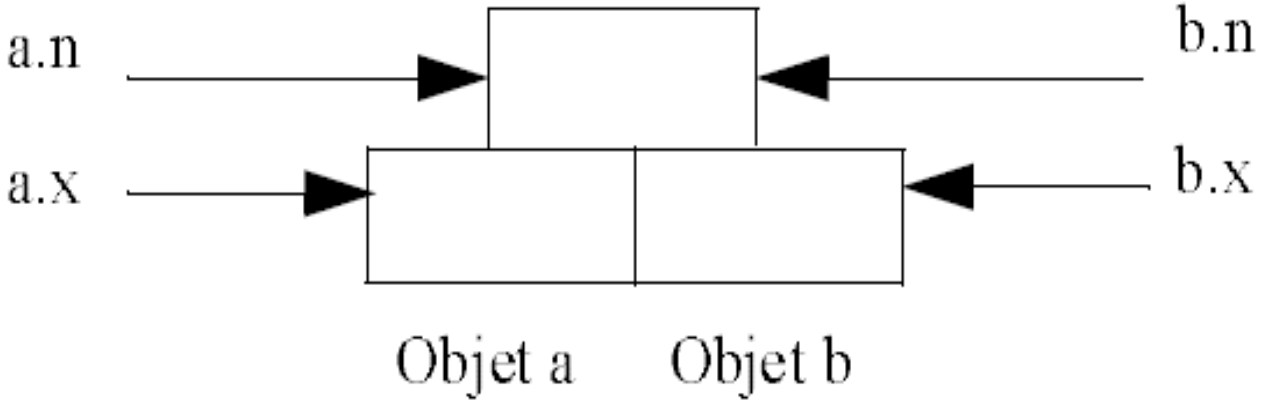
  class exple1 { int n ; float x ; ..... } ;
- Une déclaration telle que : **exple1 a, b ;** conduit à une situation que l'on peut schématiser ainsi :



LE QUALIFICATIF *STATIC* POUR UN MEMBRE DONNÉ

- Une façon de permettre à plusieurs objets de partager des données consiste à déclarer avec le qualificatif **static** les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe.
- Par exemple, si nous définissons une classe **exple2** par :

  class exple2 { static int n ; float x ; ... } ;
- La déclaration : **exple2 a, b ;** conduit à une situation que l'on peut schématiser ainsi :



INITIALISATION DES MEMBRES DONNÉES STATIQUES

- **class exple2 { static int n = 2 ; // erreur**

  ….. } ;

- En fait, cela n'est pas permis car le membre statique risquerait de se voir réserver différents emplacements dans différents modules objets.
- Un membre statique doit donc être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que : **int exple2::n = 5 ;**

INITIALISATION DES MEMBRES DONNÉES STATIQUES

Class Exemple { static int a ; float b ;

public : Exemple(float) ;

..... **} ;**

Exemple :: a=0 ; //Définition d'un membre statique nécessaire

.....

Exemple e(1) , f(2) ;

.....

- Dans la **classe Exemple**, **a** est un membre statique, **b** un membre non statique.
- Un membre statique doit être défini dans le **.cpp** tout comme une fonction membre sinon une erreur sera produite l'édition de lien.
- La définition du membre statique permet son initialisation. Ici **a = 0 ;**
- **e** et **f** sont des objets avec **e.b = 1** et **f.b = 2**
- **e** et **f** ont le membre **a** en commun **e.a = f.a = 0.**

EXEMPLE 5

- Voici un exemple de programme exploitant cette possibilité dans une classe nommée **cpte_obj**, afin de connaître, à tout moment, le nombre d'objets existants.
- Pour ce faire, nous avons déclaré avec l'attribut statique le membre **ctr**.
- Sa valeur est incrémentée de 1 à chaque appel du constructeur
- et décrémentée de 1 à chaque appel du destructeur.

EXEMPLE 5 (SUITE)

```
#include <iostream>
using namespace std ;
class cpte_obj { static int ctr ; // compteur du nombre d'objets créés
public :   cpte_obj () ;
           ~cpte_obj () ;  } ;
int cpte_obj::ctr = 0 ;  //initialisation du membre statique ctr
cpte_obj::cpte_obj () // constructeur
{ cout << "++ construction : il y a maintenant " <<
   ++ctr << " objets\n" ;  }
cpte_obj::~cpte_obj () // destructeur
{ cout << "-- destruction : il reste maintenant " << --ctr
   << "objets\n" ; }
main()  { void fct () ;    cpte_obj a ;
          fct () ;     cpte_obj b ; }
void fct () { cpte_obj u, v ; }
```

RÉSULTAT

++ construction : il y a maintenant 1 objets
++ construction : il y a maintenant 2 objets
++ construction : il y a maintenant 3 objets
-- destruction : il reste maintenant 2 objets
-- destruction : il reste maintenant 1 objets
++ construction : il y a maintenant 2 objets
-- destruction : il reste maintenant 1 objets
-- destruction : il reste maintenant 0 objets

EXPLOITATION D'UNE CLASSE

- Jusqu'ici, nous avions regroupé au sein d'un même programme trois sortes d'instructions destinées à :
  - la déclaration de la classe ;
  - la définition de la classe ;
  - l'utilisation de la classe.
- En pratique, on aura souvent intérêt à découpler la classe de son utilisation. C'est tout naturellement ce qui se produira avec une classe d'intérêt général utilisée comme un composant séparé des différentes applications.

FICHIER EN-TÊTE POUR LA CLASSE POINT

- On sera alors généralement amené à isoler les seules instructions de déclaration de la classe dans un fichier en-tête (**extension .h**) qu'il suffira d'inclure (par **#include**) pour compiler l'application.
- Par exemple, le concepteur de la classe ***point*** (définie précédemment) pourra créer le fichier en-tête suivant :

```
class point  {  /* déclaration des membres privés */
int x ;  int y ;
public :      /* déclaration des membres publics */
point (int, int) ; // constructeur
void deplace (int, int) ;
void affiche () ; } ;
```

FICHIER EN-TÊTE POUR LA CLASSE POINT (SUITE)

```
#include <iostream>
#include "point.h" // pour introduire les déclarations de la classe point
using namespace std ;
/* --- Définition des fonctions membre de la classe point --*/
point::point (int abs, int ord) { x = abs ; y = ord ; }
void point::deplace (int dx, int dy)
   { x = x + dx ; y = y + dy ; }
void point::affiche ()
{ cout << "Je suis en " << x << " " << y << "\n" ; }
```

CHAPITRE 6 : LES PROPRIÉTÉS DES FONCTIONS MEMBRES

- **Fonctions membre en ligne**
- On peut mettre une fonction en ligne mais la syntaxe est différente de celle de fonction tout court.
- Au lieu de mettre le mot clef « **inline** » devant la définition de la fonction, on écrit le corps de la fonction au même endroit que sa déclaration dans la classe.
- Ci-dessous la fonction **deplace** de la classe ***point*** est normale :

```
class point { .....
        void deplace(int dx , int dy) ;
        };
void point::deplace(int dx , int dy) { x += dx ;  y += dy;  };
```

- Par contre

FONCTIONS MEMBRE EN LIGNE

- Ci-dessous elle est **inline** bien que le mot clef ne soit pas présent.

```
Classe point { .....
        void point::deplace(int dx , int dy)
                { x += dx ;  y += dy;  }
  };
```

FONCTIONS MEMBRES STATIQUES

- **Exemple** :

```
class point { ……
        void affiche() ;
        static void affiche_tout() ;
};
```

- La fonction membre **affiche_tout** de la classe **point** est **statique**
- La fonction membre **affiche** est **normale**.

LES PROPRIÉTÉS DES FONCTIONS MEMBRES

- Nous avons déjà vu comment C++ nous autorise à **surdéfinir les fonctions ordinaires.**

- Cette possibilité s'applique également aux fonctions membres d'une classe,
  - y compris **au constructeur**
  - mais **pas au destructeur** puisqu'il ne possède pas d'argument.

- En voici un exemple, dans lequel nous surdéfinissons :

SURDÉFINITION DES FONCTIONS MEMBRES

- le constructeur **point**, le choix du bon constructeur se faisant ici suivant le nombre d'arguments :
  - 0 argument : les deux coordonnées attribuées au point construit sont toutes les deux nulles ;
  - 1 argument : il sert de valeur commune aux deux coordonnées ;
  - 2 arguments : c'est le cas « usuel » que nous avions déjà rencontré ;
- la fonction **affiche** de manière qu'on puisse l'appeler :
  - sans argument comme auparavant ;
  - avec un argument de type chaîne : dans ce cas, elle affiche le texte correspondant avant les coordonnées du point.

EXEMPLE 1

```
#include <iostream>
using namespace std ;
class point { int x, y ;
                    public : point () ;          // constructeur 1 (sans arguments)
                    point (int) ;                // constructeur 2 (un argument)
                    point (int, int) ;           // constructeur 3 (deux arguments)
                    void affiche () ;            // fonction affiche 1 (sans arguments)
                    void affiche (char *) ; // fonction affiche 2 (un argument chaîne)
   } ;
// Définition de constructeur 1  , puis  constructeur 2  et  constructeur 3
point::point () { x = 0 ; y = 0 ;  }
point::point (int abs) { x = y = abs ;  }
point::point (int abs, int ord)  { x = abs ; y = ord ; }
// Définition de fonction affiche 1  et  fonction affiche 2
void point::affiche () { cout << "Je suis en : " << x << " " << y << "\n" ;   }
void point::affiche (char * message) { cout << message ; affiche () ;  }
main()  { // appel construct 1 , affiche 1 , construct 2 , affiche 2 , construct 3 , affiche 2
point a ;   a.affiche () ;
point b (5) ;   b.affiche ("Point b - ") ;
point c (3, 12) ;  c.affiche ("Hello ---- ") ;
}
```

RÉSULTAT

Je suis en : 0 0

Point b - Je suis en : 5 5

Hello ---- Je suis en : 3 12

- Le statut **privé** ou **public** d'une fonction n'intervient pas dans les fonctions considérées.
- En revanche, si la meilleure fonction trouvée est privée, elle ne pourra pas être appelée (sauf si l'appel figure dans une autre fonction membre de la classe).

EXEMPLE (SUITE)

Considérez cet exemple :

```
class A { public : void f(int n) { ..... }
                   private : void f(char c) { ..... }  } ;
main() { int n ; char c ; A a ;  a.f(c) ; }
```

- L'appel ***a.f(c)*** amène le compilateur à considérer les deux fonctions ***f(int)*** et ***f(char)***,
- L'algorithme de recherche de la meilleure fonction conclut alors que ***f(char)*** est la meilleure fonction et qu'elle est unique.
- Mais, comme celle-ci est privée, elle ne peut pas être appelée depuis une fonction extérieure à la classe et l'appel est rejeté (et ceci, malgré l'existence de ***f(int)*** qui aurait pu convenir...).
- Rappelons que :
- • si ***f(char)*** est définie publique, elle serait bien appelée par ***a.f(c)*** ;
- • si ***f(char)*** n'est pas définie du tout, ***a.f(c)*** appellerait ***f(int)***.

AUTORÉFÉRENCE : LE MOT CLÉ *this*

- Le mot clé ***this*** utilisable uniquement au sein d'une fonction membre,
- Il désigne un pointeur sur l'objet l'ayant appelée.
- Nous proposons un exemple dans la classe ***point***, la fonction ***affiche*** fournit l'adresse de l'objet l'ayant appelée.

Exemple d'utilisation de this

```
#include <iostream>
using namespace std ;
class point { int x, y ;
public :
point (int abs=0, int ord=0)   // Un constructeur ("inline")
{ x=abs; y=ord ; }
void affiche () ;              // Une fonction affiche
  } ;
void point::affiche ()
{cout<<"Adresse:"<<this<<"- Coordonnees"<< x <<" "<< y
   <<"\n";}
main()          // Un petit programme d'essai
{ point a(5), b(3,15) ;
a.affiche ();
b.affiche ();  }
```

RÉSULTAT DE L'EXEMPLE

- **Adresse : 006AFDF0 - Coordonnees 5 0**
- **Adresse : 006AFDE8 - Coordonnees 3 15**

CHAPITRE 7 : CONSTRUCTION, DESTRUCTION ET INITIALISATION DES OBJETS

Les objets automatiques

- Les objets **automatiques** sont les objets déclarés dans une fonction ou dans un bloc. Par exemple :

```
void f() {
   Truc   t ;          // t est construit ici (dans la fonction)
    .....
      {
        point   b ;   // b est construit ici (dans le bloc)
        .....
        }             // b est détruit ici (dans le bloc)
}                     // t est détruit ici (dans la fonction)
```

- **t** et **b** sont des objets **automatiques**, **t** est visible dans la fonction **f** et **b** est dans le bloc. A la sortie de la fonction, **t** est détruit. A la sortie du bloc, **b** est détruit.

Les objets statiques

- Un objet statique est un objet déclaré avec le mot clé **static** dans une déclaration de classe ou dans une fonction ou bien à l'extérieure de toute fonction.

- Un objet statique est crée avant le début de l'exécution du programme et il est détruit à la sortie du programme.

- **Exemple 1** :
- **static** point a(1 , 7) ;

Initialisation d'un objet lors de sa déclaration

- Avec l'utilisation des constructeurs (un **constructeur crée un emplacement mémoire** pour l'objet et **l'initialise**) C++ ne permet pas de créer un objet sans l'initialiser en même temps.
- Pour créer un objet, on doit appeler un constructeur déclaré et défini ou bien le constructeur par défaut.
- Avec la déclaration de la classe suivante :

```
class point { int x , int y ;
        public : point(int abs) { x = abs ;   y = 0 ; }
        ....    };
```

- On peut écrire : point a(3) ;
- Avec la déclaration de la classe suivante :

```
struct paire { int n ; int p ; } ;
class point { ....
        point (paire  q) { x = q.n  ;   y = q.p ;   }
```

- On peut écrire : struct paire s = { 3 , 8 } ;
  point a(s) ;

Appel des constructeurs et des destructeurs

- Rappelons que si un objet possède un constructeur, sa déclaration doit obligatoirement comporter les arguments correspondants.
- Par exemple, si une classe **point** comporte le constructeur de prototype : **point (int, int),** les déclarations suivantes seront incorrectes :
  - **point a ;** // incorrect : le constructeur attend deux arguments
  - **point b (3) ;** // incorrect (même raison)

- Celle-ci, en revanche, conviendra :
  - **point a(1,7);** //correct car le constructeur possède deux arguments

APPEL DES CONSTRUCTEURS ET DES DESTRUCTEURS

- S'il existe plusieurs constructeurs, il suffit que la déclaration comporte les arguments requis par l'un d'entre eux. Ainsi, si une classe ***point*** comporte les constructeurs suivants :

  point ( ) ; // constructeur 1

  point (int, int) ; // constructeur 2

- La déclaration suivante sera rejetée :

  point a(5) ; // incorrect : aucun constructeur à un argument

- Mais celles-ci conviendront :

  point a ; // correct : appel du constructeur 1

  point b(1, 7) ; // correct : appel du constructeur 2

Exemple 2

- Un exemple de programme mettant en évidence la création et la destruction **d'objets statiques** et **automatiques**.

- Nous avons défini une classe nommée ***point***, dans laquelle le constructeur et le destructeur affichent un message permettant de repérer :

  - le moment de leur appel ;

  - l'objet concerné (nous avons fait en sorte que chaque objet de type **point** possède des valeurs différentes).

Exemple 2 (suite)

```
#include <iostream>
using namespace std ;
class point  { int x, y ;
public :   point (int abs, int ord) // constructeur ("inline")
    { x = abs ; y = ord ;
cout <<"++Construction d'un point:" << x << " "<< y <<"\n" ;
   }
~point () // destructeur ("inline")
{ cout <<"-- Destruction du point :" << x << " "<< y << "\n" ; }
   } ;
static point a(1,1) ; // un objet statique de classe point
main()   { cout << "****** Debut main *****\n" ;
point  b(10,10) ; // un objet automatique de classe point
int i ; for (i=1 ; i<=3 ; i++)
        { cout << "** Boucle tour numero " << i << "\n" ;
          point b(i,2*i) ; // objets créés dans un bloc }
cout << "****** Fin main ******\n" ;  }
```

RÉSULTAT

```
++ Construction d'un point : 1 1
****** Debut main *****
++ Construction d'un point : 10 10
** Boucle tour numero 1
++ Construction d'un point : 1 2
-- Destruction du point    : 1 2
** Boucle tour numero 2
++ Construction d'un point : 2 4
-- Destruction du point    : 2 4
** Boucle tour numero 3
++ Construction d'un point : 3 6
-- Destruction du point    : 3 6
****** Fin main ******
-- Destruction du point    : 10 10
-- Destruction du point    : 1 1
```

Les objets dynamiques

- **Cas d'une classe sans constructeur**
- Le mécanisme de gestion dynamique est donc le même que pour les structures. Ainsi, si nous définissons le type **point** suivant :

```
class point
{ int x, y ;
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche ( ) ;
} ;
```

Cas d'une classe **sans constructeur**

si nous déclarons : ***point*** *** adr ;**

- nous pourrons créer dynamiquement un emplacement de type ***point*** (qui contiendra donc ici la place pour deux entiers) et affecter son adresse à **adr** par :

  adr = *new point* ;

- L'accès aux fonctions membres de l'objet pointé par **adr** :
  adr -> initialise (1, 3) ; adr -> affiche ( ) ;
- ou, éventuellement, sans utiliser l'opérateur **->**, par :

(* adr).initialise (1, 3) ; (* adr).affiche ( ) ;

- Si l'objet contenait des membres données **publics**, on y accéderait de façon comparable.
- Quant à la suppression de l'objet en question, elle se fera, ici encore, par : ***delete*** **adr ;**

Cas d'une classe **avec constructeur**

- Après l'allocation dynamique de l'emplacement mémoire requis, l'opérateur ***new*** appellera un constructeur de l'objet.
- On voit que pour que ***new*** puisse appeler un constructeur disposant d'arguments, il est nécessaire qu'il dispose des informations correspondantes.
- En fait, elles lui seront fournies à l'aide d'une syntaxe élargie de la forme : ***new point (2, 5) ;***
- D'une manière générale, lorsque plusieurs constructeurs existent, le choix de celui qui sera appelé par ***new*** lui sera dicté par la nature des arguments figurant dans son appel.

Cas d'une classe **avec constructeur**

- Bien entendu, s'il n'existe pas de constructeur, ou s'il existe un constructeur sans argument, la syntaxe :

  new point ou ***new point ()*** sera acceptée.

- En revanche, si tous les constructeurs possèdent au moins un argument, cette syntaxe sera rejetée.

- Avant la libération de l'emplacement mémoire correspondant, l'opérateur ***delete*** appellera le destructeur.

Exemple de création dynamique d'objets

- Un exemple de programme qui crée dynamiquement un objet de type ***point*** dans la fonction **main** et qui le détruit dans une fonction **fct** (appelée par **main**).

- Les messages affichés permettent de mettre en évidence les moments auxquels sont appelés le **constructeur** et le **destructeur**.

Exemple de création dynamique d'objets (suite)

```
#include <iostream>
using namespace std ;
class point { int x, y ;
public :  point (int abs, int ord) // constructeur
{ x=abs ; y=ord ;
cout << "++ Appel Constructeur \n" ;  }
~point () // destructeur (en fait, inutile ici)
{ cout << "– Appel Destructeur \n" ;  }  } ;
main()  { void fct (point *) ; // prototype fonction fct
point * adr ;  cout << "** Debut main \n" ;
adr = new point (3,7) ; // création dynamique d'un objet
fct (adr) ;
cout << "** Fin main \n" ;  }
void fct (point * adp)  { cout << "** Debut fct \n" ;
delete adp ; // destruction de cet objet
cout << "** Fin fct \n" ;  }
```

Résultat

** Debut main
++ Appel Constructeur
** Debut fct
-- Appel Destructeur
** Fin fct
** Fin main

Le constructeur de recopie
Exemple 1 : objet transmis par valeur

- On transmettre par valeur un objet de type **vect** à une fonction ordinaire nommée **fct** qui affiche un message indiquant son appel.

```
#include <iostream>
using namespace std ;
class vect  { int nelem ;        // nombre d'éléments
              double * adr ;  // pointeur sur ces éléments
public :   vect (int n)          // constructeur "usuel"
           { adr = new double [nelem = n] ;
             cout << "+ const. usuel - adr objet : " << this
                  << " - adr vecteur : " << adr << "\n" ; }
~vect ()     // destructeur
{ cout << "- Destr. objet - adr objet : " << this << " - adr
   vecteur : " << adr << "\n" ;
delete adr ; }  } ;
void fct (vect b) { cout << "*** appel de fct ***\n" ; }
main() { vect a(5) ;  fct (a) ;  }
```

Résultat : Lorsque aucun constructeur de recopie n'a été défini

+ const. usuel - adr objet : 006AFDE4 - adr vecteur : 007D0320

***** appel de fct *****

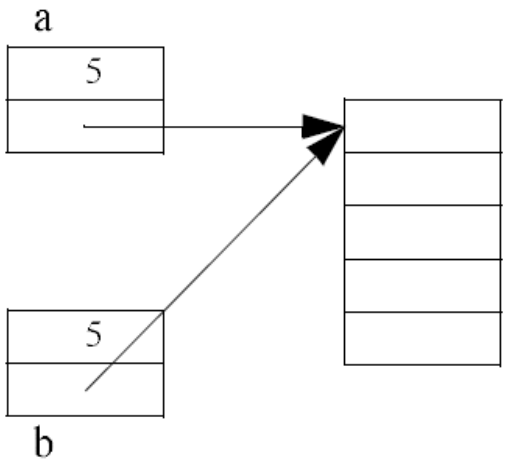
- Destr. objet - adr objet : 006AFD90 - adr vecteur : 007D0320

- Destr. objet - adr objet : 006AFDE4 - adr vecteur : 007D0320

- Comme vous pouvez le constater, l'appel : ***fct (a) ;*** a créé un nouvel objet, dans lequel on a recopié les valeurs des membres ***nelem*** et ***adr*** de ***a***.

Résultat (suite)

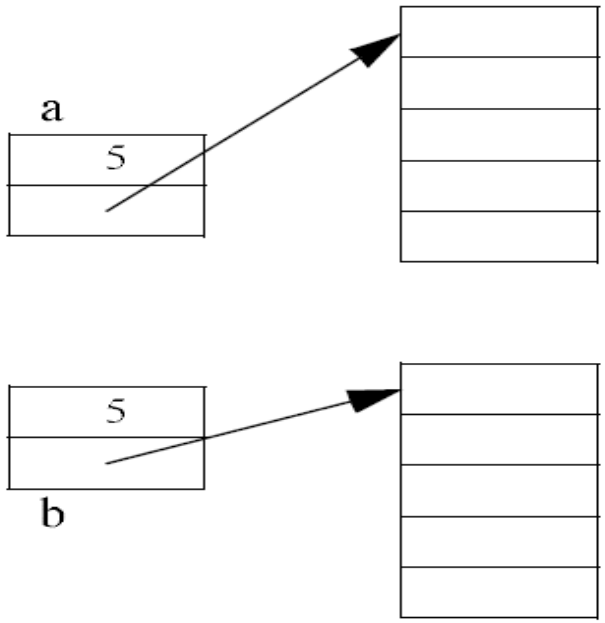
- La situation peut être schématisée ainsi (***b*** est le nouvel objet ainsi créé) :



- A la fin de l'exécution de la fonction ***fct***, le destructeur ***~vect*** est appelé pour ***b***, ce qui libère l'emplacement pointé par ***adr*** ;
- à la fin de l'exécution de la fonction ***main***, le destructeur est appelé pour ***a***, ce qui libère le même emplacement.

Définition d'un constructeur de recopie

- Pour éviter ce problème de telle sorte que l'appel: ***fct(a) ;*** conduise à créer un nouvel objet de type **vect**, avec ses membres données ***nelem*** et ***adr***,
- mais aussi son propre emplacement de stockage des valeurs du tableau. Autrement dit, nous souhaitons aboutir à cette situation :



Définition d'un constructeur de recopie (Suite)

- Pour ce faire, nous définissons, au sein de la classe ***vect***, un constructeur par recopie de la forme :

  vect (const vect &); *//ou, a la rigueur* ***vect (vect &)***

- Lors de l'appel de ***fct***. ce constructeur (appelé après la création d'un nouvel objet) doit :
  - créer dynamiquement un nouvel emplacement dans lequel il recopie les valeurs correspondant à l'objet reçu en argument ;
  - renseigner convenablement les membres données du nouvel objet
    - ***nelem*** = valeur du membre ***nelem*** de l'objet reçu en argument,
    - ***adr*** = adresse du nouvel emplacement.

Introduisons ce constructeur de recopie dans l'exemple précédent

```
#include <iostream>
using namespace std ;
class vect
 { int nelem ; // nombre d'éléments
   double * adr ; // pointeur sur ces éléments
  public :   vect (int n) // constructeur "usuel«
  { adr = new double [nelem = n] ;
   cout << "+ const. usuel - adr objet : " << this << " - adr vecteur : " << adr << "\n" ;   }
  vect (const vect & v) // constructeur de recopie
  { adr = new double [nelem = v.nelem] ; // création nouvel objet
   int i;    for(i=0;i<nelem;i++) adr[i]=v.adr[i] ; //recopie de l'ancien
 cout << "+ const. recopie - adr objet : " << this  << " - adr vecteur : " << adr << "\n" ;  }

 ~vect () // destructeur
{ cout << "- Destr. objet - adr objet : «  << this << " - adr vecteur : " << adr << "\n" ;
 delete adr ; }
  } ;

void fct (vect b)  { cout << "*** appel de fct ***\n" ; }

main() { vect a(5) ; fct (a) ;  }
```

Résultats

- + const. usuel - adr objet : 006AFDE4 - adr vecteur : 007D0320
- + const.recopie - adr objet: 006AFD88 - adr vecteur: 007D0100
- *** appel de fct ***
- - Destr. objet - adr objet : 006AFD88 - adr vecteur : 007D0100
- - Destr. objet - adr objet : 006AFDE4 - adr vecteur : 007D0320

Objets membres

- Il est tout à fait possible qu'une classe possède un membre donnée lui-même de type classe.
- Par exemple, ayant défini :

```
class point {    int x, y ;
                 public :
                    int init (int, int) ;
                   void affiche ( ) ;   } ;
```

- nous pouvons définir :

```
class cercle {   point centre ;
                   int rayon ;
                 public :
                  void affrayon ( ) ;
                  ... } ;
```

Objets membres (Suite)

- Si nous déclarons alors : ***cercle c ;***
  - l'objet ***c*** possède un membre donnée **privé** ***centre***, de type ***point***.
  - L'objet ***c*** peut accéder classiquement à la méthode ***affrayon*** par ***c.affrayon***.
- En revanche, il ne pourra pas accéder à la méthode ***init*** du membre ***centre*** car ***centre*** est **privé**.
- Si ***centre*** était **public**, on pourrait accéder aux méthodes de ***centre*** par ***c.centre.init ()*** ou ***c.centre.affiche ()***.

CHAPITRE 8 : LES FONCTIONS AMIES

- La P.O.O. pure impose l'encapsulation des données.
- Les membres **privés** (**données** ou **fonctions**) ne sont accessibles qu'aux fonctions membres (publiques ou privées) et
- seuls les membres **publics** sont accessibles « de l'extérieur ».
- En C++ « l'unité de protection » est **la classe**, c'est-à-dire qu'une même fonction membre peut accéder à tous les objets de sa classe.

Les fonctions amies (Suite)

- En revanche, ce même principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des **données privées** d'une autre classe.
- Or cette contrainte s'avère gênante dans certaines circonstances.
- Supposez par exemple que vous avez défini une classe ***vecteur*** et une classe ***matrice***.
- Vous souhaitez définir une fonction permettant de calculer le produit d'une matrice par un vecteur.

Les fonctions amies (Suite)

- Or, nous ne pourrions définir cette fonction
  - **ni** comme fonction membre de la classe ***vecteur***,
  - **ni** comme fonction membre de la classe ***matrice***,
  - et encore moins comme fonction indépendante (c'est-à-dire membre d'aucune classe).

- Vous pourriez rendre **publiques** les données de vos deux classes, mais vous perdriez alors le bénéfice de leur protection.

Les fonctions amies (Suite)

- En fait, la notion de **fonction amie** propose une solution intéressante, sous la forme d'un compromis entre **encapsulation formelle** des données **privées** et des données **publiques**.
- Lors de la définition d'une **classe**, il est en effet possible de déclarer **qu'une ou plusieurs** fonctions (extérieures à la classe) sont des « **amies** » ;
- une telle déclaration d'amitié les autorise alors à accéder aux données **privées**, au même titre que n'importe quelle **fonction membre**.

Les fonctions amies (Suite)

- Il existe plusieurs situations d'amitiés :
  - fonction indépendante, amie d'une classe ;
  - fonction membre d'une classe, amie d'une autre classe ;
  - fonction amie de plusieurs classes ;
  - toutes les fonctions membres d'une classe, amies d'une autre classe.

Exemple de fonction indépendante amie d'une classe

```
#include <iostream>
using namespace std ;
class point { int x, y ;
              public :   point (int abs=0, int ord=0) // un constructeur ("inline")
                                 { x=abs ; y=ord ; }
// Déclaration fonction amie (indépendante) nommée coincide
friend int coincide (point, point) ;  } ;

int coincide (point p, point q) // définition de coincide
{ if ((p.x == q.x) && (p.y == q.y)) return 1 ;
else return 0 ;  }

main() // programme d'essai
{ point a(1,0), b(1), c ;
if (coincide (a,b)) cout << "a coincide avec b \n" ;
else cout << "a et b sont differents \n" ;
if (coincide (a,c)) cout << "a coincide avec c \n" ;
else cout << "a et c sont differents \n" ;  }
```

Résultat de l'exemple

a coincide avec b

a et c sont differents

Fonction membre d'une classe, amie d'une autre classe

- Il s'agit un peu d'un cas particulier de la situation précédente.
- En fait, il suffit simplement de préciser, dans la déclaration d'amitié, la classe à laquelle appartient la fonction concernée, à l'aide de l'opérateur de résolution de portée (**::**).
- Par exemple, supposons que nous ayons à définir deux classes nommées ***A*** et ***B*** et que nous ayons besoin dans ***B*** d'une fonction membre ***f***, de prototype : ***int f(char, A) ;***
- ***f*** doit pouvoir accéder aux membres privés de ***A***, elle sera déclarée amie au sein de la classe par :
  friend int B::f(char, A) ;

Exemple : Fonction (*f*) membre d'une classe (B), amie d'une autre classe (A)

- *class A {*

// partie privée

.....

// partie publique

friend int B::f (char, A) ;

..... } ;

- *class B {*

.....

int f (char, A) ;

..... } ;

- *int B::f (char ..., A ...)*

{ // on a accès ici aux membres privés

// de tout objet de type A }

Fonction amie de plusieurs classes

Exemple : Fonction indépendante (f) amie de deux classes (A et B)

- *class A { // partie privée*

.....

// partie publique

friend void f(A, B) ;

..... } ;

- *class B { // partie privée*

.....

// partie publique

friend void f(A, B) ;

..... } ;

- *void f(A..., B...) { // on a accès ici aux membres privés*

// de n'importe quel objet de type A ou B }

Toutes les fonctions d'une classe B amies d'une autre classe A

- Pour dire que toutes les fonctions membres de la classe ***B*** sont amies de la classe ***A***,

- on placera, dans la classe ***A***, la déclaration :

friend class B ;

Fonction amie indépendante

Exemple : **Produit d'une matrice par un vecteur à l'aide d'une fonction indépendante amie des deux classes**

```
#include <iostream>
using namespace std ;
class matrice ; // pour pouvoir compiler la déclaration de vect
            // ********** La classe vect ********************
class vect {  double v[3] ; // vecteur à 3 composantes
                public :  vect (double v1=0, double v2=0, double v3=0) // constructeur
                                { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ;  }
friend vect prod(matrice,vect); //prod=fonction amie indépendante
void affiche () { int i ;
for (i=0 ; i<3 ; i++) cout << v[i] << " ";   cout << "\n";} } ;

// ********** La classe matrice ******************
class matrice {  double mat[3] [3] ; // matrice 3 X 3
          public :   matrice (double t[3][3]) //constructeur,à partir d'1tableau 3x3
                          { int i ; int j ;
                               for (i=0 ; i<3 ; i++)
                               for (j=0 ; j<3 ; j++)  mat[i] [j] = t[i] [j] ;  }
 friend vect prod(matrice,vect); //prod=fonction amie indépendante
} ;
```

Exemple (Suite)

```
// ********** La fonction prod ******************
vect prod (matrice m, vect x)  { int i, j ; double som ;
        vect res ; // pour le résultat du produit
for (i=0 ; i<3 ; i++) { for (j=0, som=0 ; j<3 ; j++) som +=
   m.mat[i] [j] * x.v[j] ;
res.v[i] = som ; }
return res ; }

// ********** Un petit programme de test *********
main() { vect w (1,2,3) ; vect res ;
double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
matrice a = tb ; res = prod(a, w) ; res.affiche () ; }
```

Résultat de l'exemple

14 32 50

Fonction amie, membre d'une classe

Exemple : Produit d'une matrice par un vecteur à l'aide d'une fonction membre amie d'une autre classe

```
#include <iostream>
using namespace std ;
class vect ; // pour pouvoir compiler correctement
class matrice { double mat[3] [3] ; // matrice 3 x 3
            public : matrice (double t[3][3]) //constructeur,à partir d'1tableau 3x3
              { int i ; int j ;  for (i=0 ; i<3 ; i++)  for (j=0 ; j<3 ; j++) mat[i] [j] = t[i] [j] ; }
vect prod (vect) ; // prod = fonction membre (cette fois) } ;
class vect { double v[3] ; // vecteur à 3 composantes
    public : vect (double v1=0, double v2=0, double v3=0) // constructeur
            { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
friend vect matrice::prod (vect) ; // prod = fonction amie
void affiche () { int i ; for (i=0 ; i<3 ; i++) cout << v[i] << " " ; cout << "\n" ; } } ;
// ********* Définition de la fonction prod ***********
vect matrice::prod (vect x) { int i, j ; double som ; vect res ; // pour le résultat du produit
for (i=0 ; i<3 ; i++){ for (j=0, som=0 ; j<3 ; j++)  som += mat[i] [j] * x.v[j] ;
                                                     res.v[i] = som; }
return res ;  }
main() { vect w (1,2,3) ;  vect res ;  double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
matrice a = tb ;  res = a.prod (w) ;  res.affiche () ;  }
```

Résultat de l'exemple

14 32 50

CHAPITRE 9 : L'HÉRITAGE SIMPLE

- L'héritage vous autorise à définir une nouvelle classe, dite « **dérivée** », à partir d'une classe existante dite « **de base** ».

- La classe dérivée « **héritera** » des «**potentialités** » de la classe de base, tout en lui en ajoutant de nouvelles, et cela sans qu'il soit nécessaire de remettre en question la classe de base.

La notion d'héritage

- Considérons la première ***classe point*** définie aux chapitres précédents, dont nous rappelons la déclaration:
- */* ---------- Déclaration de la classe point ---------- */*

```
class point
{ /* déclaration des membres privés */
int x ;
int y ;
/* déclaration des membres publics */
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```

La notion d'héritage

- Supposons que nous ayons besoin de définir un nouveau type classe nommé ***pointcol***, destiné à manipuler des points colorés d'un plan.
- Une telle classe peut manifestement disposer des mêmes fonctionnalités que la classe ***point***, auxquelles on pourrait adjoindre, par exemple, une méthode nommée ***colore***, chargée de définir la couleur.
- Dans ces conditions, nous pouvons être tentés de définir ***pointcol*** comme une classe dérivée de ***point***.
- Si nous prévoyons (pour l'instant) une fonction membre spécifique à ***pointcol*** nommée ***colore***, et destinée à attribuer une couleur à un point coloré,

Une classe *pointcol*, dérivée de *point*

```
class pointcol : public point // pointcol dérive de point
{ short couleur ;
public :
void colore (short cl)
        { couleur = cl ; }
} ;
```

Une classe *pointcol*, dérivée de *point*

- la déclaration ***class pointcol : public point*** spécifie que ***pointcol*** est une classe dérivée de la classe de base ***point***.
- le mot **public** signifie que les membres publics de la classe de base (***point***) seront des membres publics de la classe dérivée (***pointcol***) ;
- nous pouvons déclarer des objets de type ***pointcol*** de manière usuelle : ***pointcol p, q ;***
- Chaque objet de type ***pointcol*** peut alors faire appel :
  - aux méthodes publiques de ***pointcol*** (ici ***colore***) ;
  - aux méthodes publiques de la classe de base ***point*** (ici ***init***, ***deplace*** et ***affiche***).

Exemple d'utilisation d'une classe *pointcol*, dérivée de point

```
#include <iostream>
#include "point.h"   // incorporation des déclarations de point
using namespace std ;

/* --- Déclaration et définition de la classe pointcol ---- */
class pointcol : public point   // pointcol dérive de point
{ short couleur ;
public :  void colore (short cl) { couleur = cl ; }
}
main()
{ pointcol p ;
p.initialise (10,20) ; p.colore (5) ;
p.affiche () ;
p.deplace (2,4) ;
p.affiche () ;
}
```

Résultat de l'exemple

Je suis en 10 20

Je suis en 12 24

Utilisation des membres de la classe de base dans une classe dérivée

- La classe ***pointcol*** telle que nous l'avons définie présente des lacunes.
- Par exemple, lorsque nous appelons ***affiche*** pour un objet de type ***pointcol***, nous n'obtenons aucune information sur sa couleur.
- Une première façon d'améliorer cette situation consiste à écrire une nouvelle fonction membre publique de ***pointcol***, censée **afficher à la fois les coordonnées** et **la couleur**.
- Appelons-la pour l'instant ***affichec***.

Utilisation des membres de la classe de base dans une classe dérivée

- vous pourriez définir ***affichec*** de la manière suivante :

```
void affichec ()
{
cout << "Je suis en " << x << " " << y << "\n";
cout << " et ma couleur est : " << couleur << "\n";
}
```

Utilisation des membres de la classe de base dans une classe dérivée

- Mais alors cela signifierait que la fonction ***affichec***, membre de ***pointcol***, aurait accès aux membres privés de ***point***, ce qui serait contraire au principe d'encapsulation.
- En effet, il deviendrait alors possible d'écrire une fonction accédant directement aux données privées d'une classe, simplement en créant une classe dérivée !
- D'où la règle adoptée par C++:
- ***Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.***
- En revanche,
- **une méthode d'une classe dérivée a accès aux membres publics de sa classe de base.**

Une fonction d'affichage pour un objet de type *pointcol*

- si notre fonction membre ***affichec*** ne peut pas accéder directement aux données **privées** ***x*** et ***y*** de la classe ***point***, elle peut néanmoins faire appel à la fonction ***affiche*** de cette même classe.
- D'où une définition possible de ***affichec*** :

```
void pointcol::affichec ()
{
affiche () ;
cout << " et ma couleur est : " << couleur << "\n" ;
}
```

Une fonction d'affichage pour un objet de type ***pointcol***

- Notez bien que, au sein de ***affichec***, nous avons fait directement appel à ***affiche*** sans avoir à spécifier à quel objet cette fonction devait être appliquée.
- par convention, il s'agit de celui ayant appelé ***affichec***.
- Nous retrouvons la même règle que pour les fonctions membres d'une même classe.
- En fait, il faut désormais considérer que ***affiche*** est une fonction membre de ***pointcol***.

Une fonction d'initialisation pour un objet de type *pointcol*

- D'une manière analogue, nous pouvons définir dans ***pointcol*** une nouvelle fonction d'initialisation nommée ***initialisec***, chargée
  - d'attribuer des valeurs aux données ***x***, ***y***
  - et ***couleur***,

à partir de trois valeurs reçues en argument :

```
void pointcol::initialisec (int abs, int ord, short cl)
{
initialise (abs, ord) ;
couleur = cl ;
}
```

Nouvelle classe *pointcol* et son utilisation

```
#include <iostream>
#include "point.h" /*déclaration de la classe point (nécessaire */
                   /* pour compiler la définition de pointcol) */
using namespace std ;
class pointcol : public point { short couleur ;
public :  void colore (short cl) { couleur = cl ; }
          void affichec () ;
          void initialisec (int, int, short) ;  } ;
void pointcol::affichec ()  { affiche () ;
cout << " et ma couleur est : " << couleur << "\n" ;  }
void pointcol::initialisec (int abs, int ord, short cl)
{ initialise (abs, ord) ;
couleur = cl ;  }
main() { pointcol p ;
p.initialisec (10,20, 5) ; p.affichec () ; p.affiche () ;
p.deplace (2,4) ; p.affichec () ;
p.colore (2) ; p.affichec () ;  }
```

Résultats de l'exemple

Je suis en 10 20

et ma couleur est : 5

Je suis en 10 20

Je suis en 12 24

et ma couleur est : 5

Je suis en 12 24

et ma couleur est : 2

Redéfinition des fonctions membres d'une classe dérivée

- Dans le dernier exemple de classe ***pointcol***, nous disposions à la fois :
  - dans ***point***, d'une fonction membre nommée ***affiche*** ;
  - dans ***pointcol***, d'une fonction membre nommée ***affichec***
- Or ces deux méthodes font le même travail, à savoir afficher les valeurs des données de leur classe.
- Dans ces conditions, on pourrait souhaiter leur donner le même nom. Ceci est effectivement possible en C++,
- au sein de la fonction ***affiche*** de ***pointcol***, on ne peut plus appeler la fonction ***affiche*** de ***point*** comme auparavant :
- cela provoquerait un **appel récursif** de la fonction ***affiche*** de ***pointcol***.
- Il faut alors faire appel à l'opérateur de résolution de portée (**::**) pour localiser convenablement la méthode voulue (ici, on appellera ***point::affiche***).

Redéfinition des fonctions membres d'une classe dérivée

- Si pour un objet ***p*** de type ***pointcol***, on appelle la fonction ***p.affiche***, il s'agira de la fonction redéfinie dans ***pointcol***.

- Si l'on tient absolument à utiliser la fonction ***affiche*** de la classe ***point***, on appellera ***p.point::affiche***.

Une classe *pointcol* dans laquelle les méthodes *initialise* et *affiche* sont redéfinies

- Transformer l'exemple précédent en nommant ***affiche*** et ***initialise*** les nouvelles fonctions membres de ***pointcol***

```
#include <iostream>
#include "point.h"
using namespace std ;
class pointcol : public point { short couleur ;
public :  void colore (short cl) { couleur = cl ; }
void affiche () ; // redéfinition de affiche de point
void initialise (int,int,short); //redéfinition de initialise de point  } ;
void pointcol::affiche ()  { point::affiche () ; // appel de affiche de la classe point
cout << " et ma couleur est : " << couleur << "\n" ;  }
void pointcol::initialise (int abs, int ord, short cl)
{ point::initialise (abs,ord); //appel de initialise de classe point
couleur = cl ;  }
main()  { pointcol p ;
p.initialise (10,20, 5) ; p.affiche () ;
p.point::affiche () ; // pour forcer l'appel de affiche de point
p.deplace (2,4) ; p.affiche () ;
p.colore (2) ; p.affiche () ; }
```

Résultat de l'exemple

Je suis en 10 20

et ma couleur est : 5

Je suis en 10 20

Je suis en 12 24

et ma couleur est : 5

Je suis en 12 24

et ma couleur est : 2

Redéfinition des membres données d'une classe dérivée

- La redéfinition des fonctions membres s'applique tout aussi bien aux membres
- Si une classe ***A*** est définie ainsi :

```
class A { .....
          int a ;  char b ;
          ..... } ;
```

- une classe ***B*** dérivée de ***A*** pourra, par exemple, définir un autre membre donnée nommé ***a*** :

```
class B : public A { float a ;
                     ..... } ;
```

- Si l'objet **b** est de type ***B***, **b.a** fera référence au membre **a** de type **float** de **b**.
- Il sera toujours possible d'accéder au membre donnée **a** de type **int** (hérité de ***A***) par **b.A::a**.

- ***N.B :*** Le membre **a** défini dans ***B*** s'ajoute au membre **a** hérité de ***A*** ; il ne le remplace pas.

Appel des constructeurs et des destructeurs
La hiérarchisation des appels

- supposons que chaque classe possède un constructeur et un destructeur :

```
class A
{   .....
 public :   A (...)
            ~A ()
    .....  };

class B : public A
{   .....
 public :  B (...)
           ~B ()
    .....  };
```

La hiérarchisation des appels

- Pour créer un objet de **type *B***, il faut tout d'abord créer un objet de **type *A***, donc faire appel au constructeur de ***A***, puis le compléter par ce qui est spécifique à ***B*** et faire appel au constructeur de ***B***.
- Ce mécanisme est pris en charge par C++ **il n'y aura pas à prévoir dans le constructeur de *B* l'appel du constructeur de *A*.**:

- La même démarche s'applique aux destructeurs : lors de la destruction d'un objet de **type *B*,** il y aura automatiquement appel du destructeur de ***B***, puis appel de celui de ***A*** (les destructeurs sont appelés dans l'ordre inverse de l'appel des constructeurs).

Transmission d'informations entre constructeurs

- Un problème se pose lorsque **le constructeur de *A* nécessite des arguments.**

- En effet, les informations fournies lors de la création d'un objet de **type *B*** sont a priori destinées à son constructeur

- C++ a prévu la possibilité de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre à un constructeur de la classe de base.

Transmission d'informations entre constructeurs (suite)

- Par exemple, si l'on a ceci :

```
class point
{ .....
public :  point (int, int) ;
          ..... } ;

class pointcol : public point
{ .....
public :  pointcol (int, int, char) ;
          ..... } ;
```

- On souhaite que ***pointcol*** retransmette à ***point*** les deux premières informations reçues,

Transmission d'informations entre constructeurs (suite)

- on écrira son en-tête de cette manière :

pointcol (int abs, int ord, char cl) : point (abs, ord)

- Le compilateur mettra en place la transmission au constructeur de ***point*** des informations ***abs*** et ***ord*** correspondant (ici) aux deux premiers arguments de ***pointcol***.
- Ainsi, la déclaration : ***pointcol a (10, 15, 3) ;*** entraînera:
  - l'appel de ***point*** qui recevra les arguments ***10*** et ***15*** ;
  - l'appel de ***pointcol*** qui recevra les arguments ***10***, ***15*** et ***3***.

Transmission d'informations entre constructeurs (suite)

- En revanche, la déclaration : ***pointcol q (5, 2)*** sera rejetée par le compilateur puisqu'il n'existe aucun constructeur ***pointcol*** à deux arguments.
- Bien entendu, il reste toujours possible de mentionner des arguments par défaut dans ***pointcol***, par exemple :

pointcol (int abs =0, int ord =0, char cl =1) : point(abs,ord)

- Dans ces conditions, la déclaration : ***pointcol b (5) ;*** entraînera :
- l'appel de ***point*** avec les arguments ***5*** et ***0*** ;
- l'appel de ***pointcol*** avec les arguments ***5***, ***0*** et ***1***.

- Notez que la présence éventuelle d'arguments par défaut dans ***point*** n'a aucune incidence ici (mais on peut les avoir prévus pour les objets de type ***point***).

Exemple

```
#include <iostream>
using namespace std ;
// ********** classe point **********
class point  {  int x, y ;
public :  point (int abs=0,int ord=0)          //constructeur de point ("inline")
            { cout << "++ constr. point : " << abs << " " << ord << "\n" ;
              x = abs ; y =ord ;  }
~point ()                                      // destructeur de point ("inline")
{ cout << "-- destr. point : " << x << " " << y << "\n" ;  }   } ;
// ********** classe pointcol **********
class pointcol : public point  {   short couleur ;
          public :  pointcol (int, int,short);    //déclaration constructeur pointcol
                    ~pointcol ()                  // destructeur de pointcol ("inline")
                          { cout << "-- dest. pointcol - couleur :" << couleur << "\n" ;  }  } ;
pointcol::pointcol(int abs=0,int ord=0,short cl=1): point(abs,ord)
{ cout<<"++ constr.pointcol:"<<abs<<" "<<ord<<" "<<cl <<"\n" ;
couleur = cl ;  }
// ********** programme d'essai **********
main()   { pointcol a(10,15,3) ;     // objets
           pointcol b (2,3) ;        // automatiques
           pointcol c (12) ;         // .....
                pointcol * adr ;
adr = new pointcol (12,25) ;         // objet dynamique
delete adr ;  }
```

Résultat de l'exemple

- ++ *constr. point : 10 15*
- ++ *constr. pointcol : 10 15 3*
- ++ *constr. point : 2 3*
- ++ *constr. pointcol : 2 3 1*
- ++ *constr. point : 12 0*
- ++ *constr. pointcol : 12 0 1*
- ++ *constr. point : 12 25*
- ++ *constr. pointcol : 12 25 1*
- ……

CONTRÔLE DES ACCÈS

Les membres protégés

- Nous avons considéré deux « statuts » possibles pour un membre de classe :
- **privé** : le membre n'est accessible qu'aux fonctions membres (publiques ou privées) et aux fonctions amies de la classe ;
- **public** : le membre est accessible non seulement aux fonctions membres ou aux fonctions amies, mais également à l'utilisateur de la classe (c'est-à-dire à n'importe quel objet du type de cette classe).
- Le troisième statut – **protégé** – est défini par le mot-clé ***protected*** qui s'emploie comme les deux mots-clés précédents.
- Par exemple, la définition d'une classe peut prendre l'allure suivante :

Les membres protégés

```
class X
    { public :
      .....                  // partie publique
      protected :
      .....                  // partie protégée
      private :
      .....                  // partie privée
    };
```

- Les membres **protégés** restent inaccessibles à l'utilisateur de la classe, pour qu'ils apparaissent comme des membres **privés**.
- Mais ils seront accessibles aux membres d'une éventuelle **classe dérivée**, tout en restant dans tous les cas inaccessibles aux utilisateurs de cette classe.

Les membres protégés

- **Exemple**
- Au début nous avons évoqué l'impossibilité, pour une fonction membre d'une classe ***pointcol*** dérivée de ***point***, d'accéder aux membres **privés** ***x*** et ***y*** de ***point***.
- Si nous définissons ainsi notre classe ***point*** :

```
class point { protected : int x, y ;
        public :  point ( ... ) ;
                  affiche () ;
                  .....    } ;
```

- Il devient possible de définir, dans ***pointcol***, une fonction membre ***affiche*** de la manière suivante :

```
class pointcol : public point  {   short couleur ;
      public : void affiche ()
         { cout << "Je suis en " << x << " " << y << "\n" ;
           cout << " et ma couleur est " << couleur << "\n" ; } }
```

Dérivation publique et dérivation privée

- ***la dérivation publique***
- Les exemples précédents faisaient intervenir la forme la plus courante de dérivation, dite «**publique** » car introduite par le mot clé **public** dans la déclaration de la classe dérivée, comme dans :

  class pointcol : public point { ... } ;

  - Les membres **publics** de la classe de base sont accessibles à «tout le monde », c'est-à-dire à la fois aux fonctions membres et aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.
  - Les membres **protégés** de la classe de base sont accessibles aux fonctions membres et aux fonctions amies de la classe dérivée, mais pas aux utilisateurs de cette classe dérivée.
  - Les membres **privés** de la classe de base sont inaccessibles à la fois aux fonctions membres ou amies de la classe dérivée et aux utilisateurs de cette classe dérivée.

la dérivation publique

- De plus, tous les membres de la classe de base conservent dans la classe dérivée le statut qu'ils avaient dans la classe de base.

- Cette remarque n'intervient qu'en cas de dérivation d'une nouvelle classe de la classe dérivée

tableau récapitulant la situation

Statut dans la classe de base	Accès aux fonctions membres et amies de la classe dérivée	Accès à un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée, en cas de nouvelle dérivation
public	oui	oui	public
protégé	oui	non	protégé
privé	non	non	privé

Dérivation privée

- En utilisant le mot-clé ***private*** au lieu du mot-clé ***public***, il est possible d'interdire à un utilisateur d'une classe dérivée l'accès aux membres **publics** de sa classe de base. Par exemple :

```
class point  { .....
        public :  point (...) ;
                  void affiche () ;
                  void deplace (...) ;
                  ...  } ;

class pointcol : private point  { .....
                        public :  pointcol (...) ;
                                  void colore (...) ;
                                  ... } ;
```

Dérivation privée

- Si ***p*** est de type ***pointcol***, les appels suivants seront rejetés par le compilateur :

- ***p.affiche ()*** /* *ou même :* ***p.point::affiche ()*** */
- ***p.deplace (...)*** /* ou même : ***p.point::deplace (...)*** */

- alors que, naturellement, celui-ci sera accepté :

p.colore (...)

Les possibilités de dérivation protégée

- C++ dispose d'une possibilité supplémentaire de dérivation, dite **dérivation protégée**, intermédiaire entre la **dérivation publique** et la **dérivation privée**.
- Dans ce cas, les membres **publics** de la classe de base seront considérés comme **protégés** lors de dérivation ultérieures.
- On prendra garde à ne pas confondre le mode de dérivation d'une classe par rapport à sa classe de base (**publique**, **protégée** ou **privée**), définie par l'un des mots ***public***, ***protected*** ou ***private***, avec le statut des membres d'une classe (**public**, **protégé** ou **privé**) défini également par l'un de ces trois mots.

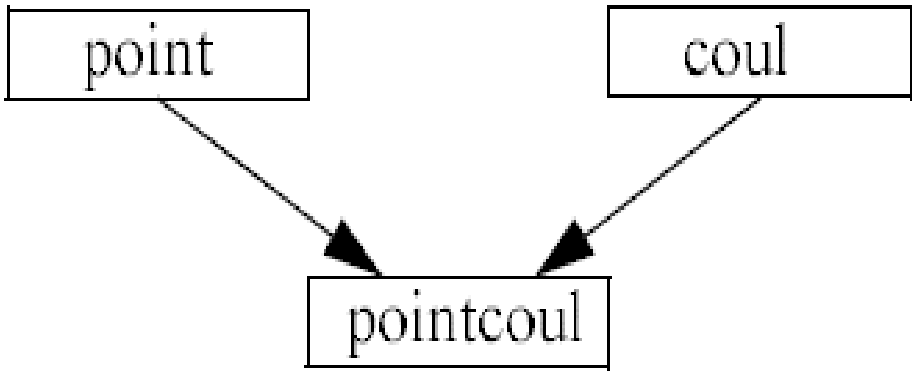
Récapitulation (Les différentes sortes de dérivations)

- La mention «**Accès FMA** » signifie : accès aux fonctions membres ou amies de la classe ;
- La mention «**nouveau statut** » signifie : statut qu'aura ce membre dans une éventuelle classe dérivée).

Classe de base			Dérivée publique		Dérivée protégée		Dérivée privée	
Statut ini-tial	Accès FMA	Accès utilisateur	Nouveau statut	Accès utili-sateur	Nouveau statut	Accès utili-sateur	Nouveau statut	Accès utilisateur
public	O	O	public	O	protégé	N	privé	N
protégé	O	N	protégé	N	protég	N	privé	N
privé	O	N	privé	N	privé	N	privé	N

Héritage Multiple

- Considérons une situation simple, celle où une classe, que nous nommerons **pointcoul**, hérite de deux autres classes nommées **point** et **coul** :



Héritage Multiple

- Supposons que les classes **point** et **coul** se présentent ainsi :

```
class point
   { int x, y ;
public :
point (...) {...}
~point () {...}
affiche () {...}
} ;
```

```
class coul
    { short couleur ;
public :
    coul (...) {...}
    ~coul () {...}
    affiche () {...}
    } ;
```

- Nous pouvons définir une classe **pointcoul** héritant de ces deux classes en la déclarant ainsi (ici, nous avons choisi **public** pour les deux classes, mais nous pourrions employer **private** ou **protected**) :

```
class pointcoul : public point, public coul
        { ... } ;
```

Héritage Multiple

- Notez que nous nous sommes contentés de remplacer la mention d'une classe de base par une liste de mentions de classes de base.
- Dans le cas de l'héritage simple, le constructeur devait pouvoir retransmettre des informations au constructeur de la classe de base.
- Il en va de même ici, avec cette différence qu'il y a deux classes de base. L'en-tête du constructeur se présente ainsi :

pointcoul (.....) : point (.....), coul (.....)

arguments de **pointcoul**	arguments à transmettre à **point**	arguments à transmettre à **coul**

- L'ordre d'appel des constructeurs est le suivant :
  - constructeurs des classes de base, dans l'ordre où les classes de base sont déclarées dans la classe dérivée (ici, **point** puis **coul**) ;
  - constructeur de la classe dérivée (ici, **pointcoul**).
- Les destructeurs éventuels seront, là encore, appelés dans l'ordre inverse lors de la destruction d'un objet de type **pointcoul**.

Héritage Multiple

- Lorsque plusieurs fonctions membres portent le même nom dans différentes classes, on peut lever l'ambiguïté en employant **l'opérateur de résolution de portée**.
- Ainsi, la fonction affiche de **pointcoul** sera :

  void affiche () { point::affiche () ; coul::affiche () ; **}**
- Bien entendu, si les fonctions d'affichage de **point** et de **coul** se nommaient par exemple **affp** et **affc**, la fonction **affiche** aurait pu s'écrire simplement :

  void affiche () { affp () ; affc () ; **}**
- Un objet de type **pointcoul** peut faire appel aux **fonctions membres** de **pointcoul**, ou éventuellement aux **fonctions membres des classes de base point** et **coul** (en se servant de **l'opérateur de résolution de portée** pour lever des ambiguïtés).

Un exemple d'héritage multiple : pointcoul hérite de point et de coul

```
#include <iostream>
using namespace std ;
class point  { int x, y ;
public :  point (int abs, int ord) { cout << "++ Constr. point \n" ; x=abs ; y=ord ; }
~point () { cout << "-- Destr. point \n" ; }
void affiche () { cout << "Coordonnees : " << x << " " << y << "\n" ; }   } ;
class coul { short couleur ;
public :coul (int cl)  { cout << "++ Constr. coul \n" ; couleur = cl ;  }
~coul () { cout << "-- Destr. coul \n" ; }
void affiche ()  { cout << "Couleur : " << couleur << "\n" ;  }   } ;
class pointcoul : public point, public coul   { public :   pointcoul (int, int, int) ;
~pointcoul () { cout << "---- Destr. pointcoul \n" ; }
void affiche ()  { point::affiche () ; coul::affiche () ;  }   } ;
pointcoul::pointcoul (int abs, int ord, int cl) : point (abs, ord), coul (cl)
{ cout << "++++ Constr. pointcoul \n" ;  }
main()  { pointcoul p(3,9,2) ;
cout << "-----------\n" ;   p.affiche () ;          // appel de affiche de pointcoul
cout << "-----------\n" ;   p.point::affiche () ; // on force l'appel de affiche de point
cout << "-----------\n" ;   p.coul::affiche () ;   // on force l'appel de affiche de coul
cout << "-----------\n" ;  }
```

RÉSULTATS

```
++ Constr. point
++ Constr. coul
++++ Constr. pointcoul
-----------
Coordonnees : 3 9
Couleur : 2
-----------
Coordonnees : 3 9
-----------
Couleur : 2
-----------
---- Destr. pointcoul
-- Destr. coul
-- Destr. point
```

CHAPITRE 10 :LES FONCTIONS VIRTUELLES ET LE POLYMORPHISME

- ***situation où le typage dynamique est nécessaire***

class point { void affiche () ;

..... } ;

class pointcol : public point { void affiche () ;

..... } ;

point p ; pointcol pc ; point * adp = &p ;

- L'instruction : ***adp -> affiche () ;*** appelle la méthode ***affiche*** du type ***point***.
- Mais si nous exécutons cette affectation (autorisée) : ***adp=&pc;*** le pointeur ***adp*** pointe maintenant sur un objet de type ***pointcol***.

Situation où le typage dynamique est nécessaire

- Néanmoins, l'instruction : ***adp ->affiche() ;*** fait toujours appel à la méthode ***affiche*** du type ***point***,
- alors que le type ***pointcol*** dispose lui aussi d'une méthode ***affiche***.
- En effet, le choix de la méthode à appeler a été réalisé lors de la compilation ;
- il a donc été fait en fonction du type de la variable ***adp***. C'est la raison pour laquelle on parle de « **ligature statique** ».

Le mécanisme des fonctions virtuelles

- C++ va nous permettre de faire en sorte que l'instruction: ***adp-> affiche ()*** appelle non plus systématiquement la méthode ***affiche*** de ***point***,
- mais celle correspondant au type de l'objet réellement désigné par ***adp*** (ici ***point*** ou ***pointcol***).
- Pour ce faire, il suffit de déclarer « **virtuelle** » (mot-clé ***virtual***) la méthode ***affiche*** de la classe ***point*** :

```
class point  { .....
                    virtual void affiche () ;
                    ..... } ;
```

- Cette instruction indique au compilateur que les éventuels appels de la fonction **affiche** doivent utiliser une **ligature dynamique** et non plus une **ligature statique**.

Le mécanisme des fonctions virtuelles

- Autrement dit, lorsque le compilateur rencontrera un appel tel que : ***adp -> affiche () ;*** il ne décidera pas de la procédure à appeler.
- Il effectue le choix de la fonction qu'au moment de l'exécution de cette instruction.
- Ce choix étant basé sur le type exact de l'objet ayant effectué l'appel (plusieurs exécutions de cette même instruction pouvant appeler des fonctions différentes).

Le mécanisme des fonctions virtuelles

- Dans la classe ***pointcol***, on ne procédera à aucune modification : **il n'est pas nécessaire de déclarer virtuelle, dans les classes dérivées,** une fonction déclarée **virtuelle** dans une classe de base.

- A titre d'exemple, voici le programme dans lequel nous nous sommes contentés de rendre **virtuelle** la fonction ***affiche*** :

Mise en œuvre d'une ligature dynamique (ici pour *affiche*) par la technique des fonctions virtuelles

```
#include <iostream>
using namespace std ;
class point  { protected : int x, y ;   //pour que x et y soient accessibles à pointcol
public :  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
virtual void affiche () { cout << "Je suis un point \n" ;
cout << " mes coordonnees sont : " << x << " " << y << "\n" ; }    } ;
class pointcol : public point  { short couleur ;
   public :  pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
                                        { couleur = cl ;     }
void affiche ()   { cout << "Je suis un point colore \n" ;
cout << " mes coordonnees sont : " << x << " " << y ;
cout << " et ma couleur est : " << couleur << "\n" ;  }    } ;
main()  { point p(3,5) ; point * adp = &p ;
pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
adp->affiche () ; adpc->affiche () ;
cout << "---------------\n" ;
adp = adpc ;                          // adpc = adp serait rejeté
adp->affiche () ;
adpc->affiche () ;  }
```

Résultats

Je suis un point
mes coordonnées sont : 3 5
Je suis un point colore
mes coordonnées sont : 8 6 et ma couleur est : 2

Je suis un point colore
mes coordonnées sont : 8 6 et ma couleur est : 2
Je suis un point colore
mes coordonnées sont : 8 6 et ma couleur est : 2

la ligature dynamique est indispensable

- Dans l'exemple précédent, lors de la conception de la classe ***point***, nous avons prévu que chacune de ses descendantes redéfinirait à sa guise la fonction ***affiche***.
- Cela conduit à prévoir, dans chaque fonction, des instructions d'affichage des coordonnées.
- Pour éviter cette redondance, nous pouvons définir la fonction ***affiche*** (de la classe ***point***) de manière qu'elle :
  - affiche les coordonnées (action commune à toutes les classes) ;
  - faire appel à une autre fonction (nommée par exemple ***identifie***), ayant pour vocation d'afficher les informations spécifiques à chaque objet.
  - Nous supposons que chaque descendante de ***point*** redéfinira ***identifie*** de façon appropriée (mais elle n'aura plus à prendre en charge l'affichage des coordonnées).

Redéfinir la classe ***point***

```
class point  { int x, y ;
public :  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
          void identifie ()
         { cout << "Je suis un point \n" ; }
          void affiche ()
        {   identifie () ;
cout<<"Mes coordonnees sont : " << x << " " << y<<"\n" ;
}
```

Dérivons une classe ***pointcol*** en redéfinissant la fonction ***identifie***

```
class pointcol : public point { short couleur ;
public :
pointcol(int abs=0,int ord=0, int cl=1) : point (abs, ord)
        { couleur = cl ; }
void identifie ()
{ cout << "Je suis un point colore de couleur : " <<
  couleur << "\n" ; }
  } ;
```

- Si nous cherchons alors à utiliser ***pointcol*** de la façon suivante:

```
pointcol pc (8, 6, 2) ;
pc.affiche () ;
```

Résultat et interprétation

- Nous obtenons le résultat :

  Je suis un point

  Mes coordonnées sont : 8 6

- ce qui n'est pas ce que nous espérions !

- Certes, la compilation de l'appel : ***pc.affiche ()*** a conduit le compilateur à appeler la fonction ***affiche*** de la classe ***point*** (puisque cette fonction n'est pas redéfinie dans ***pointcol***).
- En revanche, à ce moment-là, l'appel : ***identifie ()*** figurant dans cette fonction a déjà été compilé en un appel… d'identifie de la classe ***point***.

problème de ligature statique

- Bien qu'ici la fonction ***affiche*** ait été appelée explicitement pour un objet (et non, comme précédemment, à l'aide d'un pointeur), nous nous trouvons à nouveau en présence d'un problème de ligature statique.
- Pour le résoudre, il suffit de déclarer **virtuelle** la fonction **identifie** dans la classe ***point***.
- Cela permet au compilateur de mettre en place les instructions assurant l'appel de la fonction ***identifie*** correspondant au type de l'objet l'ayant effectivement appelée.
- La situation est légèrement différente de celle qui nous a servi à présenter les fonctions **virtuelles**.
- En effet, l'appel d'***identifie*** est réalisé non plus directement par l'objet lui-même, mais indirectement par la fonction ***affiche***.

Exemple

- Un programme complet reprenant les définitions des classes ***point*** et ***pointcol***.
- Il montre comment un appel tel que ***pc.affiche ()*** entraîne bien l'appel de ***identifie*** du type ***pointcol***.
- A titre indicatif, nous avons introduit quelques appels par pointeur, afin de montrer que, là aussi, les choses se déroulent convenablement.

Exemple Complet

```
#include <iostream>
using namespace std ;
class point { int x, y ;
  public :  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
virtual void identifie () { cout << "Je suis un point \n" ; }
void affiche ()  { identifie () ;
cout << "Mes coordonnees sont : " << x << " " << y << "\n" ; }  } ;
class pointcol : public point  { short couleur ;
public : pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
{ couleur = cl ; }
void identifie () { cout << "Je suis un point colore de couleur : " <<
   couleur << "\n" ; }  } ;
main() { point p(3,4) ; pointcol pc(5,9,5) ;
p.affiche () ; pc.affiche () ; cout << "----------\n";
point * adp = &p ; pointcol * adpc = &pc ;
adp->affiche () ; adpc->affiche () ; cout << "----------\n";
adp = adpc ;
adp->affiche () ; adpc->affiche () ;
}
```

Résultat

- *Je suis un point*
- *Mes coordonnees sont : 3 4*
- *Je suis un point colore de couleur : 5*
- *Mes coordonnees sont : 5 9*
- *----------*
- *Je suis un point*
- *Mes coordonnees sont : 3 4*
- *Je suis un point colore de couleur : 5*
- *Mes coordonnees sont : 5 9*
- *----------*
- *Je suis un point colore de couleur : 5*
- *Mes coordonnees sont : 5 9*
- *Je suis un point colore de couleur : 5*
- *Mes coordonnees sont : 5 9*

Quelques restrictions et conseils

- **Seule une fonction membre peut être virtuelle**

- Cela se justifie par le mécanisme employé pour effectuer la ligature dynamique, à savoir un choix basé sur le type de l'objet ayant appelé la fonction.

- Cela ne pourrait pas s'appliquer à une fonction «**ordinaire** » (même si elle était amie d'une classe).

- **Un constructeur ne peut pas être virtuel**

- Un **constructeur** ne peut être appelé que pour un type classe parfaitement défini qui sert, précisément, à définir le type de l'objet à construire.

- A priori, donc, un constructeur n'a aucune raison d'être soumis au polymorphisme.

- D'ailleurs, on peut penser qu'on n'appelle jamais un constructeur par pointeur ou référence.

- **Un destructeur peut être virtuel**

- En revanche, un destructeur peut être virtuel.

- Il est toutefois conseillé de prendre quelques précautions à ce sujet.

- Dans une classe de base (destinée à être dérivée), prévoir :
  - soit **aucun destructeur** ;
  - soit un destructeur **privé** ou **protégé** ;
  - soit un destructeur **public et virtuel**.