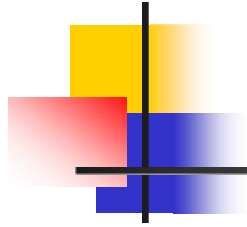


Constructeur de copie
Fonctions Amies
Membres statiques d'une classe
Surdéfinition des opérateurs
Héritage



P00

Constructeur de recopie



Exemple 1: constructeur de recopie par défaut

```
class Time
{ int Hour, Minute, Second;
  public :
    Time(){}
    Time(int H,int M, int S){Hour=H; Minute=M; Second=S;}
    void Afficher(){...}
};
```

```
void main()
{
    Time T1(10,12,15);
    Time T2(T1);
    T2.Afficher();
}
```



Constructeur de recopie

- ❑ Un constructeur de recopie est un constructeur qui permet de créer un objet et de l'initialiser avec le contenu d'un autre objet de la même classe.
- ❑ Comme son nom l'indique, le constructeur de recopie est spécialisé dans la création d'un objet à partir d'un autre objet pris comme modèle.
- ❑ Toute classe dispose d'un constructeur de copie par défaut généré automatiquement par le compilateur, dont le seul but est de recopier les champs de l'objet à recopier un à un dans les champs de l'objet à instancier.
- ❑ Toutefois, ce constructeur par défaut ne suffira pas toujours, et le programmeur devra parfois en fournir un **explicitement**.



Constructeur de recopie

- ❑ Ce sera notamment le cas lorsque certaines données des objets auront été allouées **dynamiquement**. Une copie brutale des champs d'un objet dans un autre ne ferait que recopier les pointeurs, **pas** les données pointées.
- ❑ *Ainsi, la modification de ces données pour un objet entraînerait la modification des données de l'autre objet, ce qui ne serait sans doute pas l'effet désiré.*
- ❑ La définition des constructeurs de copie se fait comme celle des constructeurs normaux. *Le nom doit être celui de la classe*, et il ne doit y avoir aucun type. Dans la liste des paramètres cependant, il devra toujours y avoir une référence sur l'objet à copier :



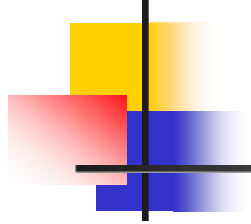
Exemple 2

```
class chaine { char * s;  
    public: int taille;  
    chaine () {.....}  
    chaine(const chaine &Source);  
    ~chaine(void);  
};
```

```
chaine::chaine(const chaine &Source)  
{  
    int i = 0; // Compteur de caractères.  
    Taille = Source.Taille;  
    s = new char[Taille+1] //effectue l'allocation.  
    strcpy(s, Source.s) ; // Recopie la chaîne de caractères source.  
    return;  
}
```

Le constructeur de copie est appelé dans toute instantiation avec initialisation, comme celles qui suivent :

```
chaine s2(s1); // s1 objet déjà existant  
chaine s2=s1;
```



P00

Fonctions Amies



Fonctions Amies

- ❑ En C++, l'unité de protection est la classe, et non pas l'objet. Cela signifie qu'une fonction membre d'une classe peut accéder à tous les membres privés de n'importe quel objet de sa classe.
- ❑ En revanche, ces membres privés restent inaccessibles à n'importe quelle fonction membre d'une autre classe ou à n'importe quelle fonction indépendante.
- ❑ La notion de **fonction amie**, ou plus exactement de « **déclaration d'amitié** », permet de déclarer dans une classe les fonctions que l'on autorise à accéder à ses membres privés (données ou fonctions). Il existe plusieurs situations d'amitié
- ❑ Une fonction amie d'une classe est une fonction qui, bien que non-membre de la classe, a accès aux données privées de la classe.
- ❑ L'amitié est déclarée en utilisant le mot-clé réservé: **friend**.
- ❑ L'amitié n'est ni symétrique ni transitive



Fonctions Amies

1) Fonction indépendante, amie d'une classe A

```
class A
{
.....
friend fct (-----);
.....
}
```

La fonction fct ayant le prototype spécifié est autorisée à accéder aux membres privés de la classe A.

2) Fonction membre d'une classe B, amie d'une autre classe A

```
class A
{
.....
friend B:fct (-----);
.....
};
```

La fonction fct, membre de la classe B, ayant le prototype spécifié, est autorisée à accéder aux privés de A



Fonctions Amies

2) Fonction membre d'une classe B, amie d'une autre classe A (suite)

- ❑ Pour qu'il puisse compiler convenablement la déclaration de A, donc en particulier la déclaration d'amitié relative à **fct**, le compilateur devra connaître la déclaration de B (*mais pas nécessairement sa définition* !).
- ❑ En revanche, pour compiler *la définition de fct*, le compilateur devra posséder les caractéristiques de A, donc disposer de sa déclaration.



Fonctions Amies

3) Toutes les fonctions d'une classe B sont amies d'une autre classe A

- ❑ Dans ce cas, plutôt que d'utiliser autant de déclarations d'amitié que de fonctions membre, on utilise (dans la déclaration de la classe A) la déclaration (globale) suivante :

```
class A
{
.....
friend class B ;
.....
};
```



Application

- ❑ Soit une classe vecteur 3d ayant 3 données membres privées, de type entier, les composantes du vecteur (x,y,z) . Elle a un constructeur permettant d'initialiser les données membres.
- ❑ Ecrire une fonction indépendante, comparer, amie de la classe vecteur3d, permettant de savoir si 2 vecteurs ont mêmes composantes.
- ❑ Si $v1$ et $v2$ désignent 2 vecteurs de type vecteur3d, écrire le programme qui permet de tester l'égalité de ces 2 vecteurs.



Solution

```
#include <iostream.h>
```

```
class vecteur3d{
```

```
    int x;
```

```
    int y;
```

```
    int z;
```

```
public:
```

```
    vecteur3d(int a=0,int b=0, int c=0);
```

```
    friend void comparer (vecteur3d p, vecteur3d q);
```

```
};
```

```
vecteur3d::vecteur3d(int a,int b,int c){
```

```
    x=a;
```

```
    y=b;
```

```
    z=c; }
```

```
void comparer(vecteur3d p, vecteur3d q)
{
    if (p.x==q.x && p.y==q.y && p.z==q.z)
    { cout<<"2 vecteurs égaux"<<endl;}
    else
        cout<<« Désolé !!!"<<endl;
```

```
void main(){
    vecteur3d v1(3,2,5);
    vecteur3d v2(3,4,5);
    comparer(v1,v2); }
```



POO

Membres statiques d'une Classe



Attributs statiques

- ❑ L'une des caractéristiques les plus importantes des classes est que **chaque instance (objet) possède sa propre copie d'attributs** (*sa propre zone mémoire*). Cette copie constitue l'identité propre à l'objet.
- ❑ La durée de vie des attributs est égale dans ce cas à la durée de vie de l'objet.
- ❑ Toutefois, dans certains cas, il est intéressant de créer **des attributs qui sont indépendants des objets** et qui sont plutôt liés à la classe ou en d'autres mots au modèle que définit la classe.
- ❑ De tels attributs peuvent être partagées par toutes les instances de la classe et sont appelés **des attributs statiques**.



Déclaration d'un attribut statique

La déclaration d'un attribut statique se fait à l'intérieur de la définition de la classe à l'aide du mot-clé *static* comme suit :

```
class NomClasse  
{ ... ..  
    static Type NomAttributStatique;  
    ... ..  
};
```




Définition d'un Attribut statique

- ❑ Un attribut statique doit être **obligatoirement défini**. Cette définition se fait d'une manière globale en dehors de la classe sans l'utilisation du mot-clé ***static***. Toutefois le nom de l'attribut doit être complètement qualifié comme suit :

Type NomClasse::NomAttributStatique = Valeur;

- ❑ L'affectation d'une valeur initiale à l'attribut statique lors de sa définition n'est pas obligatoire. Un attribut statique non explicitement initialisé sera automatiquement initialisé par le système à 0.



Exemple 1

- ☐ Supposons que l'on souhaite créer par programmation une scène qui représente une chute de neige.
- ☐ Cette neige sera représentée par un ensemble de cercles de même rayon remplis par la couleur blanche.
- ☐ Le programme devra également donner la possibilité à l'utilisateur de modifier la taille des grains de neige. Cette modification affectera l'attribut *Rayon* de tous les cercles de manière uniforme.



Exemple 1 (suite)

- ❑ Une première possibilité pour représenter les cercles consiste à utiliser la classe suivante :

```
❑ class Cercle
{   int x;
    int y;
    int Rayon;
    public :
        Cercle(int xi, int yi){x=xi; y=yi;}
        Cercle(){x=0; y=0;}
        void Deplacer(int dx, int dy){x+=dx; y+=dy;}
        void SetRayon(int R){Rayon = R;}
        void Dessiner() {...}
};
```

- ❑ Avec cette spécification chaque cercle possède sa propre copie de l'attribut **Rayon**.

Exemple 1 (suite)

- ❑ Or puisque tous les cercles possèdent la même taille, il devient plus intéressant de faire partager l'attribut *Rayon* par toutes les instances de cette classe en le déclarant comme statique.

- ❑ `class Cercle`

- `{`

- `int x;`

- `int y;`

- `static int Rayon;`

- `public :`

- `Cercle(int xi, int yi){x=xi; y=yi;}`

- `Cercle(){x=0; y=0;}`

- `void Deplacer(int dx, int dy){x+=dx; y+=dy;}`

- `void SetRayon(int R){Rayon = R;}`

- `void Dessiner(){...}`

- `};`

```
// Définition de l'attribut statique
// en dehors de la classe
int Cercle::rayon=2;
```



Caractéristiques

- ❑ Il permet un gain en espace mémoire. En effet, une seule zone mémoire sera utilisée par toutes les instances.
- ❑ Il permet de diminuer la quantité de code nécessaire à la modification d'une caractéristique commune à toutes les instances.
 - ❑ Par exemple, pour modifier la taille de la neige, il suffit de modifier la valeur de l'attribut *Rayon* une seule fois. Cette modification affectera toutes les instances.



Caractéristiques

- ❑ Un attribut statique est partagé par toutes les instances d'une classe. Sa durée de vie ne dépend pas de celles des objets.
- ❑ Un attribut statique existe en mémoire même si aucun objet de la classe n'a encore été créé.
- ❑ Un attribut statique est souvent appelé **attribut de classe** (son existence dépend de l'existence de la classe).
- ❑ Un attribut non statique est appelé **attribut d'instance** (son existence est liée à l'instanciation de la classe).
- ❑ Un attribut statique peut être manipulé par les méthodes de sa classe comme n'importe quel autre attribut non statique.



Exercice

- ❑ Supposons que l'on souhaite calculer l'espace mémoire occupé par toutes les instances d'une classe dans un programme en cours d'exécution. Ce calcul passe par la détermination du nombre d'objets instanciés.
- ❑ Une solution possible à ce problème consiste à utiliser un attribut statique qui s'incrémente à chaque instanciation d'un nouvel objet et qui se décrémente à chaque destruction d'un objet.
 - **Proposer une implémentation de cette solution avec la classe *Point* qui représente un point dans un repère à deux dimensions.**

Solution

```
class Point  
{
```

```
    int x, y;
```

```
    public:
```

```
        static int Memory;
```

```
        Static int nb;
```

```
        Point(int a, int b):x(a),y(b)
```

```
        {    Memory+=sizeof(Point);
```

```
        }    Nb++;
```

```
        ~Point()
```

```
        {Memory-=sizeof(Point);
```

```
        Nb--;}  
};
```

```
int Point::Memory = 0;
```

```
Int Point::nb=0;
```

```
void main()
```

```
{
```

```
    cout<<" Mémoire occupée :"<<Point::Memory<<endl;
```

```
    Point P1(10,10);
```

```
    cout<<" Mémoire occupée :"<<Point::Memory<<endl;
```

```
    Point *P2= new Point(5,5);
```

```
    cout<<" Mémoire occupée :"<<Point::Memory<<endl;
```

```
    delete P2;
```

```
Point(int a, int b)
```

```
{x=a; y=b;
```

```
Memory+=sizeof(Point);
```

```
Nb++;
```

```
}
```




P00

Surcharge des opérateurs



Le mécanisme de la surdéfinition d'opérateurs

- ❑ **Surdéfinir** les opérateurs = **leur donner une nouvelle signification** lorsqu'ils portent (en partie ou en totalité) sur des objets de type classe

- ❑ Pour surdéfinir un opérateur existant *op*, on définit une fonction nommée **operator op** :
 - ❑ soit sous forme d'une fonction indépendante (généralement amie d'une ou de plusieurs classes) ;
 - ❑ soit sous forme d'une fonction membre d'une classe. Dans le premier cas, si *op* est un opérateur binaire, la notation *a op b* est équivalente à :
operator op (a, b)



Les possibilités et les limites de la surdéfinition d'opérateurs

- ❑ On doit se limiter aux opérateurs existants, en **conservant leur « pluralité »** (unaire, binaire). Les opérateurs ainsi surdéfinis gardent leur priorité et leur associativité habituelle
- ❑ Un opérateur surdéfini doit toujours posséder un opérande de type classe (on ne peut donc pas modifier les significations des opérateurs usuels).
- ❑ Les opérateurs **[]**, **()**, **->**, **new** et **delete** doivent obligatoirement être définis comme fonctions membre.



Cas particuliers

- ❑ Les opérateurs `=` (affectation) et `&` (pointeur sur) possèdent une signification prédéfinie pour les objets de n'importe quel type classe.
- ❑ Cela ne les empêche nullement d'être surdéfinis.
- ❑ La surdéfinition de `new`, pour un type classe donné, se fait par une fonction de prototype : `void * new (size_t)`
- ❑ Elle reçoit, en unique argument, la taille de l'objet à allouer (cet argument sera généré automatiquement par le compilateur, lors d'un appel de `new`), et elle doit fournir en retour l'adresse de l'objet alloué.



Application

Soit une classe vecteur3d définie comme suit :

```
class vecteur3d
{ float x, y, z ;
  public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
      { x = c1 ; y = c2 ; z = c3 ; }
};
```

Définir les opérateurs == et != de manière qu'ils permettent de tester la coïncidence ou la non-coïncidence de deux points

- a. en utilisant des fonctions membres ?
- B. en utilisant les fonctions amies ?

Solution

```
class vecteur3d
{ float x, y, z ;
public :
```

```
    vecteur3d (float c1=0.0, float c2=0.0, float
c3=0.0)
```

```
    { x = c1 ; y = c2 ; z = c3 ;}
```

```
    int operator == (vecteur3d) ;
```

```
    int operator != (vecteur3d) ;
```

```
};
```

```
int vecteur3d::operator == (vecteur3d v)
```

```
{ if ( (v.x == x) && (v.y == y) && (v.z ==z) )
    return 1 ;
```

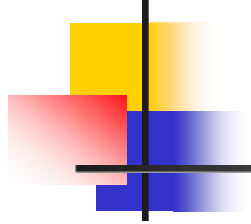
```
    else return 0 ;
```

```
}
```

```
int vecteur3d::operator != (vecteur3d v)
```

```
{ return ! ( (*this) == v ) ;}
```

Solution (suite)



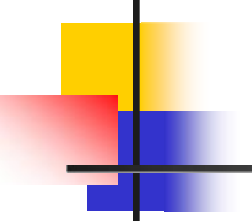
```
#include <iostream>  
using namespace std ;
```

```
main()  
{  
    vecteur3d v1 (3,4,5), v2 (4,5,6), v3 (3,4,5) ;  
    cout << "v1==v2 : " << (v1==v2) << " v1!=v2 : " << (v1!=v2) << "\n" ;  
    cout << "v1==v3 : " << (v1==v3) << " v1!=v3 : " << (v1!=v3) << "\n" ;  
}
```

//Exécution

```
v1==v2 : 0 v1!=v2 : 1  
v1==v3 : 1 v1!=v3 : 0
```

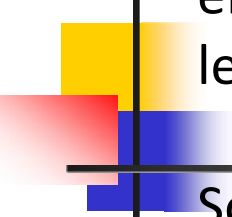
Solution avec fonctions amies



```
class vecteur3d
{ float x, y, z ;
  public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
    { x = c1 ; y = c2 ; z = c3 ; }
    friend int operator == (vecteur3d, vecteur3d) ;
    friend int operator != (vecteur3d, vecteur3d) ;
} ;
```

```
int operator == (vecteur3d v, vecteur3d w)
{ if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) )
  return 1 ;
else return 0 ; }
```

```
int operator != (vecteur3d v, vecteur3d w)
{ return ! ( v == w ) ; }
```

L'objectif de cet exercice est de définir les opérateurs arithmétiques en C++ d'une classe **Complexe** en utilisant les fonctions membres et les fonctions amies.

Soit la classe **complexe** pour gérer les nombres complexes:

Attributs:

Re la partie réelle de type **double**

Img la partie imaginaire de type **double**

Méthodes:

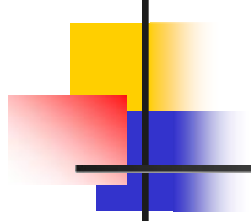
- **Constructeur** avec arguments (valeurs d'initialisation par default)
- Fonction d'**affichage**
- Fonction qui retourne le **module** ($z = \sqrt{\text{Re}^2 + \text{Img}^2}$)
- Fonction qui retourne le **conjugué** ($\text{Re} - \text{Img} * i$)

Surcharge des opérateurs suivants :

"+" : (complexe + complexe)

De même pour **"*", "-"**

Créer un programme de test **main ()**.



P00

L'Héritage



Définition & intérêts

Définition

- C'est le fait de définir une nouvelle classe en se basant sur la définition d'une **classe déjà existante**. La nouvelle classe **hérite** alors les attributs et les méthodes de la classe existante et ce en plus de ses propres membres.
- L'héritage est un concept **propre à la POO**.

Remarques

- La classe dérivée constitue une **spécialisation** de la classe de base.
- La classe de base constitue une **généralisation** de la classe dérivée.

Intérêts

- Réutilisation des modules déjà existants.
- Factorisation des propriétés communes à un certain nombre de classes →
Rendre les programmes moins complexes et plus facilement maintenables.

Exemple

LIVRE

Reference
Titre
Auteur
Editeur

GetReference()
GetTitre()
GetAuteur()
GetEditeur()

PERIODIQUE

Reference
Titre
DirecteurRedaction
Numero

GetReference()
GetTitre()
GetDirecteurRedaction()
GetNuméro()



DOCUMENT

Reference
Titre

GetReference()
GetTitre()

LIVRE

Auteur
Editeur

GetAuteur()
GetEditeur()

PERIODIQUE

DirecteurRedaction
Numero

GetDirecteurRedaction()
GetNuméro()



Terminologie utilisée par le concept d'héritage

- La nouvelle classe définie par héritage est appelée ***classe dérivée*** ou ***classe fille***.
- La classe à partir de laquelle se fait l'héritage est appelée ***classe de base, super classe*** ou également ***classe mère***.
- L'héritage est également appelé ***dérivation de classes***.
- La relation d'héritage est schématisée graphiquement par une flèche qui part de la classe dérivée vers la classe de base.
- Cette relation peut s'exprimer par la phrase "***est un***".



Syntaxe de dérivation

```
class NomClasseDérivée : [Mode_Dérivation] NomClasseDeBase  
{  
    Liste des nouveaux membres de la classe dérivée  
};
```

- ***NomClasseDérivée*** : désigne la nouvelle classe créée par dérivation.
- ***NomClasseDeBase*** : désigne le nom de la classe à partir de laquelle se fait la dérivation.
- ***Mode_Dérivation*** : sert à indiquer le mode avec lequel sera faite la dérivation. Ce mode détermine les droits d'accès qui seront attribués aux membres de la classe de base dans la classe dérivée. Les modes qui sont définis en C++ sont : *public*, *protected* et *private*.



Exemple

```
class Document
{
    char Reference[10];
    char Titre[100];
    void Afficher();
    char* GetReference();
    char* GetTitre();
};
```

```
class Livre : public Document
{
    private:
        char Auteur[10];
        char Editeur[100];
    public:
        char* GetAuteur();
        char* GetEditeur();
};
```



Modalités d'accès à la classe de base

Les membres privés d'une classe de base ne sont jamais accessibles aux fonctions membre de sa classe dérivée. Outre les « statuts » public ou privé, il existe un statut « **protégé** ».

Un ***membre protégé*** se comporte comme un membre privé pour un utilisateur quelconque de la classe, mais comme un membre public pour la classe dérivée.



Exemple

```
class A
{
    private int x1;
    protected :
        int x2;
    public :
        int x3;
};
```

```
void main()
{
    A Obj1;
```

```
    Obj1.x1 = 5; // Erreur car x2 est privé dans A
    Obj1.x2 = 1; // Erreur car x2 est protégé dans A
    Obj2.x3 = 8; // OK
```

```
}
```



Modalités d'accès à la classe de base : Mode de dérivation

■ **publique (public)** – les membres de la classe de base conservent leur statut dans la classe dérivée ; c'est la situation la plus usuelle ;

■ **privée (private)** – tous les membres de la classe de base deviennent privés dans la classe dérivée ;

■ **protégée (protected)** – les membres publics de la classe de base deviennent membres protégés de la classe dérivée ; les autres membres conservent leur statut.



Exemple

```
class B : public A
{
    private : int b1, b2; protected : void f()
        { b1 = a1*2; //erreur a1 :privé dans A !
        }
    void g()
        { b2 = a2*a3;
          //OK car a2 et a3 sont accessibles dans B
        }
};
```

```
class A
{
    int a1;
    protected :
        int a2;
    public :
        int a3;
};
```

```
void main()
{
    A Obj1;
    B Obj2;
    Obj1.a2 = 6; // Erreur car
a2 est protégé dans A
    Obj2.a3 = 8; // OK
    Obj2.a2=10; //OK
    Objet2.a1 = 20; //NON
}
```



Exercice :

On dispose d'un fichier nommé point.h contenant la déclaration suivante de la classe point :

```
class point  
{ float x, y ;  
public :  
void initialise (float abs=0.0, float ord=0.0)  
{ x = abs ; y = ord ;  
}  
void affiche ()  
{ cout << "Point de coordonnées : " << x << " " << y << "\n" ; }  
float abs () { return x ; }  
float ord () { return y ; }  
};
```

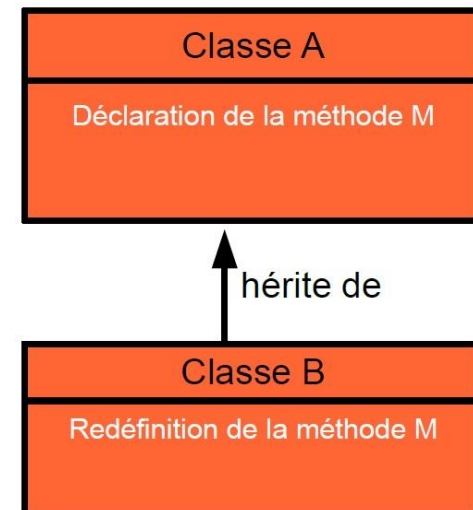


Exercice :

- a. Créer une classe `pointb`, dérivée de `point` comportant simplement une nouvelle fonction membre nommée `rho`, fournissant la valeur du rayon vecteur (première coordonnée polaire) d'un point.
- b. Même question, en supposant que les membres `x` et `y` ont été déclarés protégés (`protected`) dans `point`, et non plus privés.
- c. Introduire un constructeur dans la classe `pointb`.
- d. Quelles sont les fonctions membre utilisables pour un objet de type `pointb` ?

Commentaires:

- Si la **classe fille** possède un **constructeur**, celui-ci ne sera **appelé** qu'une fois le **constructeur de la classe mère** exécuté. Dans le cas des destructeurs, l'ordre **est** inverse, celui de la **classe fille est appelé** avant celui de la **classe mère**.
- Si la classe fille ne définit pas de constructeur, le constructeur par défaut de la classe mère est appelé ?
- La **redéfinition de méthode** (ou **overriding** en anglais) est une notion liée à l'héritage: c'est le fait de pouvoir redéfinir dans la classe fille une méthode déjà définie dans la classe mère.





Exercice :

On dispose de la déclaration de la classe cercle :

```
class cercle
{ int x, y ;
  Int rayon ;
public :
cercle (int abs=0, int ord=0, int R=0)
{ x = abs ; y = ord ; rayon=R ;}
void affiche ()
{ cout << "Coordonnées : " << x << " " << y << <<rayon"\n" ; }
void deplace (int dx, int dy)
{ x = x + dx ; y = y + dy ; }
};
```



Exercice (suite):

- a. Créer une classe `cercle2`, dérivée de `cercle`, comportant :
- un membre donnée supplémentaire `cl`, de type `int`, contenant la « couleur » d'un cercle ;
 - les fonctions membre suivantes :
 - `affiche` (redéfinie), qui affiche les coordonnées et la couleur d'un cercle de type `cercle2` ;
 - `colore` (`int couleur`), qui permet de définir la couleur d'un objet de type `cercle2`,
 - un constructeur permettant de définir la couleur et les coordonnées (on ne le définira pas en ligne).

b. Que fera alors précisément cette instruction : `cercle2 (2, 3, 5) ;`