POO sous C++ - TP3

Objectifs

Les conversions de type définies par l'utilisateur, L'héritage simple

Exercice 1

Soit la classe **point** suivante :

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

a. La munir d'un opérateur de cast permettant de convertir un point en un entier (correspondant à son abscisse).
b. Soient alors ces déclarations :

```
point p ;
int n ;
void fct (int) ;
```

Que font ces instructions :

```
n = p ; // instruction 1
fct (p) ; // instruction 2
```

Exercice 2

Quels résultats fournira le programme suivant :

```
#include <iostream>
using namespace std ;
class point{
int x, y ;
public :
point (int abs, int ord){// constructeur 2 arguments
x = abs ; y = ord ;
}
operator int(){  // "cast" point --> int
cout << "*** appel int() pour le point " << x << " " << y << "\n" ;
return x ;
}
};
main(){
point a(1,5), b(2,8) ;
int n1, n2, n3 ;
n1 = a + 3 ; // instruction 1
cout << "n1 = " << n1 << "\n" ;
n2 = a + b ; // instruction 2
cout << "n2 = " << n2 << "\n" ;
```

```
double z1, z2 ;
z1 = a + 3 ; // instruction 3
cout << "z1 = " << z1 << "\n" ;
z2 = a + b ; // instruction 4
cout << "z2 = " << z2 << "\n" ;
}
```

Introduire une fonction membre nommée *coincide* permettant de savoir si deux vecteurs ont les mêmes composantes :

a. en utilisant une transmission par valeur ;
b. en utilisant une transmission par adresse ;
c. en utilisant une transmission par référence.

Si *v1* et *v2* désignent 2 vecteurs de type *vecteur3d*, comment s'écrit le test de coïncidence de ces 2 vecteurs, dans chacun des 3 cas considérés ?

Exercice 3

Que se passerait-il si, dans la classe point du précédent exercice, nous avions introduit, en plus de l'opérateur operator int, un autre opérateur de cast operator double ?

Exercice 4

Considérer la classe suivante :

```
class complexe{
double x, y ;
public :
complexe (double r=0, double i=0) ;
complexe (complexe &) ; // ou complexe (const complexe &)
} ;
```

Dans un programme contenant les déclarations :

```
complexe z (1,3) ;
void fct (complexe) ;
```

que produiront les instructions suivantes :

```
z = 3.75 ;  // instruction 1
fct (2.8) ; // instruction 2
z = 2 ;  // instruction 3
fct (4) ;  // instruction 4
```

Exercice 5

a. Quels résultats fournira le programme suivant :

```
#include <iostream>
using namespace std ;
class point{
int x, y ;
public :
point (int abs=0, int ord=0){ // constructeur 0, 1 ou 2 arguments
x = abs ; y = ord ;
cout << "$$ construction point : " << x << " " << y << "\n" ;
}
friend point operator + (point, point) ; // point + point --> point
void affiche (){
cout << "Coordonnées : " << x << " " << y << "\n" ;
}
} ;
```

```
point operator+ (point a, point b){
point r ;
r.x = a.x + b.x ; r.y = a.y + b.y ;
return r ;
}
main(){
point a, b(2,4) ;
a = b + 6 ; // affectation 1
a.affiche() ;
a = 6 + b ; // affectation 2
b.affiche() ;
}
```

b. Même question en supposant que l'opérateur + a été surdéfini comme une fonction membre et non plus comme une fonction amie. **c.** Même question en supposant que l'opérateur + est défini ainsi :

```
friend point operator + (point &, point &) ;
```

d. Même question en supposant que l'opérateur + est défini ainsi :

```
friend point operator + (const point &, const point &) ;
```

Exercice 6
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 ; }
};
```

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** ?

Exercice 7
On dispose d'un fichier **point.h** contenant la déclaration suivante de la classe point :

```
#include <iostream>
using namespace std ;
class point{
float x, y ;
public :
point (float abs=0.0, float ord=0.0){
x = abs ; y = ord ;
}
void affiche (){
cout << "Coordonnées : " << x << " " << y << "\n" ;
```

```
}
void deplace (float dx, float dy){
x = x + dx ; y = y + dy ;
}
} ;
```

a. Créer une classe pointcol, dérivée de point, comportant :
• un membre donnée supplémentaire cl, de type int, contenant la "couleur" d'un point ;
• les fonctions membre suivantes :

`affiche (redéfinie), qui affiche les coordonnées et la couleur d'un objet de type pointcol ;`
`colore (int couleur), qui permet de définir la couleur d'un objet de type pointcol,`

un constructeur permettant de définir la couleur et les coordonnées (on ne le définira pas en ligne).
b. Que fera alors précisément cette instruction :

```
pointcol (2.5, 3.25, 5) ;
```

Exercice 8
On suppose qu'on dispose de la même classe **point** (et donc du fichier **point.h**) que dans l'exercice précédent. Créer une classe **pointcol** possédant les mêmes caractéristiques que ci-dessus, mais sans faire appel à l'héritage. Quelles différences apparaîtront entre cette classe **pointcol** et celle de l'exercice précédent, au niveau des possibilités d'utilisation ?

Exercice 9
Soit une classe point ainsi définie (nous ne fournissons pas la définition de son constructeur) :

```
class point{
int x, y ;
public :
point (int = 0, int = 0) ;
friend int operator == (point, point) ;
} ;
int operator == (point a, point b){
return a.x == b.x && a.y == b.y ;
}
```

Soit la classe **pointcol**, dérivée de **point** :

```
class pointcol : public point {
int cl ;
public :
pointcol (int = 0, int = 0, int = 0) ;
// éventuelles fonctions membre
} ;
```

a. Si a et b sont de type **pointcol** et **p** de type **point**, les instructions suivantes sont-elles correctes et, si oui, que font-elles ?

```
if (a == b) ... // instruction 1
if (a == p) ... // instruction 2
if (p == a) ... // instruction 3
if (a == 5) ... // instruction 4
if (5 == a) ... // instruction 5
```

b. Mêmes questions, en supposant, cette fois, que l'opérateur + a été défini au sein de **point**, sous forme d'une fonction membre.

Exercice 10
Soit une classe **vect** permettant de manipuler des "vecteurs dynamiques" d'entiers (c'est-à-dire dont la dimension peut être fixée au moment de l'exécution) dont la déclaration (fournie dans le fichier **vect.h**) se présente ainsi :

```
class vect{
int nelem ;  // nombre d'éléments
int * adr ;  // adresse zone dynamique contenant les éléments
public :
vect (int) ;  // constructeur (précise la taille du vecteur)
~vect () ;  // destructeur
int & operator [] (int) ; // accès à un élément par son "indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire pour le nombre d'entiers reçu en argument et que l'opérateur [] peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.
Créer une classe **vectb**, dérivée de **vect**, permettant de manipuler des vecteurs dynamiques, dans lesquels on peut fixer les " bornes " des indices, lesquelles seront fournies au constructeur de **vectb**. La classe **vect** apparaîtra ainsi comme un cas particulier de **vectb** (un objet de type **vect** étant un objet de type **vectb** dans lequel la limite inférieure de l'indice est 0).
On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type **vectb**.

Exercice 11
Soit une classe **int2d** permettant de manipuler des "tableaux dynamiques" d'entiers à deux dimensions dont la déclaration (fournie dans le fichier **int2d.h**) se présente ainsi :

```
class int2d{
int nlig ; // nombre de "lignes"
int ncol ; // nombre de "colonnes"
int * adv ; // adresse emplacement dynamique contenant les valeurs
public :
int2d (int nl, int nc) ; // constructeur
~int2d () ;   // destructeur
int & operator () (int, int) ; // accès à un élément, par ses 2 "indices"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur [] peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.
Créer une classe **int2db**, dérivée de **int2d**, permettant de manipuler des tableaux dynamiques, dans lesquels on peut fixer les " bornes " (valeur minimale et valeur maximale) des deux indices ; les quatre valeurs correspondantes seront fournies en arguments du constructeur de **int2db**.
On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type **int2db**.

Exercice 12
Soit une classe vect permettant de manipuler des " vecteurs dynamiques " d'entiers, dont la déclaration (fournie dans un fichier **vect.h**) se présente ainsi (notez la présence de membres protégés) :

```
class vect{
protected : // en prévision d'une éventuelle classe dérivée
int nelem ; // nombre d'éléments
int * adr ; // adresse zone dynamique contenant les éléments
public :
vect (int) ; // constructeur
~vect () ; // destructeur
int & operator [] (int) ; // accès à un élément par son "indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur [] peut être utilisé indifféremment dans une expression ou à gauche d'une affectation. En revanche, comme on peut le

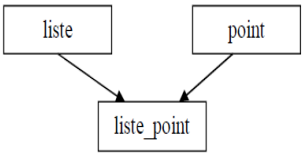
voir, cette classe n'a pas prévu de constructeur par recopie et elle n'a pas surdéfini l'opérateur d'affectation. L'affectation et la transmission par valeur d'objets de type vect posent donc les " problèmes habituels ".
Créer une classe **vectok**, dérivée de **vect**, telle que l'affectation et la transmission par valeur d'objets de type **vectok** s'y déroulent convenablement. Pour faciliter l'utilisation de cette nouvelle classe, introduire une fonction membre **taille** fournissant la dimension d'un vecteur.
Écrire un petit programme d'essai.

Exercice 13
Soit la classe point créée dans l'exercice 10, dont la déclaration était la suivante :

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

Adapter cette classe, de manière que la fonction membre *affiche* fournisse, en plus des coordonnées du point, le nombre d'objets de type *point*.

Exercice 14
Interpréter et commenter le programme utilisant l'héritage multiple dans l'exemple suivant de liste simplement chaînée de points (objets appartenant à la classe 'point').




```
struct element{
    element *suivant;
    void *contenu;
};
//----------------------------------------
class liste{
    element *debut;
    element *courant;
    public:
        liste(){
            debut=NULL;
            courant=debut;
        }
        ~liste();
        void ajouter();
        void *premier(){
            courant=debut;
            return(courant->contenu);
        }
        void *prochain(){
            if(courant!=NULL) courant=courant->suivant;
            return(courant->contenu);
        }
        int finir(){
            return(courant==NULL);
        }
};
```