# Corrigé Série 5 : Programmation C++ (Les classes-Heritage)

## Exercice 1 :

Voici les résultats que fournit le programme (ici, nous avons utilisé le compilateur C++ Builder X ; ce point n'ayant d'importance que pour les objets temporaires, dans le cas des implémentations qui ne respecteraient pas la norme quant à l'instant de leur destruction).
La plupart des lignes sont assorties d'explications complémentaires présentées conventionnellement sous forme de commentaires et qui précisent les instructions concernées.

```
début main
constructeur I             : 1 0   /* demo a ;           */
constructeur I             : 2 0   /* demo b = 2 ;       */
constructeur II (recopie) : 1 0   /* demo c = a ;       */
constructeur I             : 3 3   /* new demo (3, 3)    */
constructeur II (recopie) : 1 0   /* recopie de la valeur de a dans fct(a, ...)
*/
                                  /*   ce qui crée un objet temporaire
*/
entrée fct
destruction                : 3 3   /* delete add ;   (dans fct)
*/
sortie fct
destruction                : 1 0   /* destruction objet temporaire créé pour
*/
                                  /*  l'appel de fct
*/
constructeur I             : 4 4   /* demo d = demo(4, 4)
*/
constructeur I             : 5 5   /* c = demo(5, 5) (contruction objet temporaire)
*/
destruction                : 5 5   /* destruction objet temporaire précédent
*/
fin main
destruction                : 4 4   /* destruction d */
destruction                : 5 5   /* destruction c */
destruction                : 2 0   /* destruction b */
destruction                : 1 0   /* destruction a */
```

Notez bien que l'affectation `c = demo (5,5)` entraîne la création d'un objet temporaire par appel du constructeur de `demo` (arguments `5` et `5`) ; cet objet est ensuite affecté à `a`. On constate d'ailleurs que cet objet est effectivement détruit aussitôt après. Mais il existe certaines implémentations qui ne respectent pas la norme et où cela peut se produire plus tard.
Par ailleurs, l'appel de `fct` a entraîné la construction d'un objet temporaire, par appel du constructeur par recopie. Cet objet est ici libéré dès la sortie de la fonction. Là encore, dans certaines implémentations, cela peut se produire plus tard.

## Exercice 2 :

Pour pouvoir numéroter nos points, il nous faut pouvoir compter le nombre de fois où le constructeur a été appelé, ce qui nous permettra bien d'attribuer un numéro différent à chaque point.
Pour ce faire, nous définissons, au sein de la classe `point`, un membre donnée statique `nb_points`. Ici, il sera incrémenté par le constructeur mais le destructeur n'aura pas d'action sur lui. Comme tout membre statique, `nb_points` devra être initialisé.
Voici la déclaration (définition) de notre classe, accompagnée du programme d'utilisation demandé :

```
#include <iostream>
using namespace std ;
class point
{   int num;
    static int nb_points ;
  public :
    point ()
       {  num = ++nb_points ;
          cout << "création point numéro    : " << num << "\n" ;
       }
```

```
    ~point ()
       {  cout << "Destruction point numéro : " << num << "\n" ;
       }
} ;
int point::nb_points=0 ;     // initialisation obligatoire
main()
{  point * adcourb = new point [4] ;
   delete [] adcourb ;
}
```

## Exercice 3 :

1. La déclaration de la classe découle de l'énoncé :

```
        /*         fichier SETINT1.H        */
        /* déclaration de la classe set_int */
class set_int
{
   int * adval ;               // adresse du tableau des valeurs
   int nmax ;                  // nombre maxi d'éléments
   int nelem ;                 // nombre courant d'éléments

public :
   set_int (int = 20) ;        // constructeur
   ~set_int () ;               // destructeur
   void ajoute (int) ;         // ajout d'un élément
   int appartient (int) ;      // appartenance d'un élément
   int cardinal () ;           // cardinal de l'ensemble
} ;
```

Le membre donnée `adval` est destiné à pointer sur le tableau d'entiers qui sera alloué par le constructeur. Le membre `nmax` représentera la taille de ce tableau, tandis que `nelem` fournira le nombre effectif d'entiers stockés dans ce tableau. Ces entiers seront, cette fois, rangés dans l'ordre où ils seront fournis à `ajoute`, et non plus à un emplacement prédéterminé, comme nous l'avions fait pour les caractères (dans les exercices du chapitre précédent).

Comme la création d'un objet entraîne ici une allocation dynamique d'un emplacement mémoire, il est raisonnable de prévoir la libération de cet emplacement lors de la destruction de l'objet ; cette opération doit donc être prise en charge par le destructeur, d'où la présence de cette fonction membre.

Voici la définition de notre classe :

```
#include "setint1.h"
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de valeurs
   nelem = 0 ;
}
set_int::~set_int ()
{  delete adval ;                    // libération tableau de valeurs
}
void set_int::ajoute (int nb)
{   // on examine si nb appartient déjà à l'ensemble
    //   en utilisant la fonction membre appartient
    // s'il n'y appartient pas et si l'ensemble n'est pas plein
    //   on l'ajoute
   if (!appartient (nb) && (nelem<nmax))  adval [nelem++] = nb ;
}
int set_int::appartient (int nb)
{  int i=0 ;
     // on examine si nb appartient déjà à l'ensemble
     // (si ce n'est pas le cas, i vaudra nele en fin de boucle)
   while ( (i<nelem) && (adval[i] != nb) ) i++ ;
   return (i<nelem) ;
}
int set_int::cardinal ()
{  return nelem ;
}
```

Notez que, dans la fonction membre `ajoute`, nous avons utilisé la fonction membre `appartient` pour vérifier que le nombre à ajouter ne figurait pas déjà dans notre ensemble.

Par ailleurs, l'énoncé ne prévoit rien pour le cas où l'on cherche à ajouter un élément à un ensemble déjà « plein » ; ici, nous nous sommes contentés de ne rien faire dans ce cas. Dans la pratique, il faudrait soit prévoir un moyen

pour que l'utilisateur soit prévenu de cette situation, soit, mieux, prévoir automatiquement l'agrandissement de la zone dynamique associée à l'ensemble.

2. Voici le programme d'utilisation demandé :

```
#include "setint1.h"
#include <iostream>
using namespace std ;
main()
{  set_int ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
      { cin >> n ;
        ens.ajoute (n) ;
      }
   cout << "il y a : " << ens.cardinal () << " entiers différents\n" ;
}
```

À titre indicatif, voici un exemple d'exécution :

```
donnez 20 entiers
0 2 5 2 8 5 1 8 2 0 7 2 5 5 4 5 0 0 4 5
il y a : 7 entiers différents
```

3. Telle qu'est actuellement prévue notre classe `set_int`, si un objet de ce type est transmis par valeur, soit en argument d'appel, soit en retour d'une fonction, il y a appel du constructeur de recopie par défaut. Or ce dernier se contente d'effectuer une copie des membres donnée de l'objet concerné, ce qui signifie qu'on se retrouve en présence de deux objets contenant deux pointeurs différents sur un même tableau d'entiers. Un problème va donc se poser, dès lors que l'objet copié sera détruit (ce qui peut se produire dès la sortie de la fonction) ; en effet, dans ce cas, le tableau dynamique d'entiers sera détruit, alors même que l'objet d'origine continuera à « pointer » dessus. Indépendamment de cela, d'autres problèmes similaires pourraient se poser si la fonction était amenée à modifier le contenu de l'ensemble ; en effet, on modifierait alors le tableau d'entiers original, chose à laquelle on ne s'attend pas dans le cas de la transmission par valeur.

Pour régler ces problèmes, il est nécessaire de munir notre classe d'un constructeur par recopie approprié, c'est-à-dire tenant compte de la « partie dynamique » de l'objet (on parle parfois de « copie profonde »). Pour ce faire, on alloue un second emplacement pour un tableau d'entiers, dans lequel on recopie les valeurs du premier ensemble. Naturellement, il ne faut pas oublier de procéder également à la recopie des membres donnée, puisque celle-ci n'est plus assurée par le constructeur de recopie par défaut (lequel n'est plus appelé, dès lors qu'un constructeur par recopie a été défini).

Nous ajouterons donc dans la déclaration de notre classe :

```
set_int (set_int &) ;          // constructeur par recopie
```

Et, dans sa définition :

```
set_int::set_int (set_int & e)  // ou set_int::set_int (const set_int & e)
{
   adval = new int [nmax = e.nmax] ; // allocation nouveau tableau
   nelem = e.nelem ;
   int i ;
   for (i=0 ; i<nelem ; i++)   // copie ancien tableau dans nouveau
   adval[i] = e.adval[i] ;
```

## Exercice 4 :

a. La fonction `colore` ne pose aucun problème particulier puisqu'elle agit uniquement sur un membre donnée propre à `pointcool`. En ce qui concerne `affiche`, il est nécessaire qu'elle puisse afficher les valeurs des membres `x` et `y`, hérités de `point`. Comme ces membres sont privés (et non protégés), il n'est pas possible que la nouvelle méthode `affiche` de `pointcol` accède à eux directement. Elle doit donc obligatoirement faire appel à la méthode `affiche` du type `point` ; il suffit, pour cela, d'utiliser l'opérateur de résolution de portée. Enfin, le constructeur de `pointcol` doit retransmettre au constructeur de `point` les coordonnées qu'il aura reçues par ses deux premiers arguments.

Voici ce que pourrait être notre classe `pointcol` (ici, toutes les fonctions membre, sauf le constructeur, sont en ligne) :

```
         /****** fichier pointcol.h :déclaration de pointcol ******/
#include "point.h"
#include <iostream>
using namespace std ;
class pointcol : public point
{   int cl ;
  public :
    pointcol (float = 0.0, float = 0.0, int = 0) ;
    void colore (int coul)
      { cl = coul ;
      }
    void affiche ()                    // affiche doit appeler affiche de
      { point::affiche () ;            // point pour les coordonnées
        cout << "  couleur : " << cl ; // mais elle a accès à la couleur
      }
} ;
         /****** définition du constructeur de pointcol *****/
#include "point.h"
#include "pointcol.h"
pointcol::pointcol (float abs, float ord, int coul) : point (abs, ord)
{  cl = coul ;       // on pourrait aussi écrire colore (coul) ;
}
```

Notez bien que l'on précise le constructeur de `point` devant être appelé par celui de `pointcol`, au niveau du constructeur de `pointcol`, et non de sa déclaration.

b. La déclaration `pointcol (2.5, 3.25, 5)` entraîne la création d'un emplacement pour un objet de type `pointcol`, lequel est initialisé par appel, successivement :

- du constructeur de `point`, qui reçoit en argument les valeurs `2.5` et `3.25` (comme prévu dans l'en-tête du constructeur de `pointcol`)þ;
- du constructeur de `pointcol`.