1**) Vecteurs numériques et vecteurs creux**

Soit, dans un espace de dimension N de grande taille (que l'on fixera à 1000), un vecteur $X = \{X_1, X_2, \ldots, X_N\}$, dont les valeurs X_i sont des entiers. Un vecteur creux permet de représenter un petit nombre, noté pn(X), des éléments du vecteur X dont les valeurs ne sont pas nulles : $pn(X) = card \{i \in [1, N]; X_i \neq 0\}$.

Ce type de situation arrive fréquemment en calcul scientifique : météo, mécanique des fluides, etc. Dans ce cas, on veut que les calculs de z = x + y ou de z = K * x prennent de l'ordre de pn(x + y) ou K * pn(x) opérations, soit quelques dizaines au lieu d'un millier d'additions (voire bien plus si N >> 1000) ou de multiplications d'éléments presque tous nuls.

Pour fixer les idées, considérons les vecteurs v1, v2 et v3 dont les seuls éléments non nuls sont ainsi définis :

v1[1] = 13	v2[1] = -9	v3[0] = 11
v1[518] = -2	v2[3] = -5	v3[3] = -612
v1[611] = 117	v2[6] = 28	v3[5] = 403
v1[[817] = -312		
v1[942] = 177		

Le vecteur des valeurs utiles de v1 est ici vut(v1) = {13, -2, 117, -312, 177}.

De façon analogue aux matrices ou aux polygones creux, on ne cherche pour les vecteurs creux et par souci d'efficacité à ne mémoriser que les éléments non nuls de ces vecteurs. Les indices de ces éléments peuvent ainsi être réduits à un nouveau vecteur, appelé vecteur d'index, et défini ainsi :

$vix(X) = \{card\{i \in [0, N - 1]; Xi \neq 0\}\}$

Sur l'exemple considéré, on a vix(v1) = {1, 518, 611, 817, 942} ; vix(v1) peut être vu comme un index des indices des éléments non nuls de v1.

On cherche à mémoriser les vecteurs `vut` ET `vix` au travers d'une classe `Vecteur` définie comme suit :

```
class Vecteur {
public :
        Vecteur (int tval[], int ne)            // constructeur de base, fourni
        {
                nbe = ne ;
                ptab = new int[nbe];
                for (int i = 0 ; i < ne ; i++)
                        ptab[i] = tval[i];
        }
        Vecteur (int n) ;                       // alloue à ptab un tableau de n entiers et
                                                // initialise tous ses éléments à 0 (1.1)
        Vecteur (const Vecteur& v);             // constructeur par copie (1.2)
        ~Vecteur ();                            // destructeur (1.3)
        Vecteur& operator= (const Vecteur&)     // surcharge de l'affectation (1.4)
        int get_val (int indice) const;         // retourne ptab[indice] (1.5a)
        void set_val(int indice, int val);      // exécute ptab[indice] = val; (1.5b)
        int taille () const;                    // retourne nbe (1.5c)
        friend ostream& operator<< (ostream&, const Vecteur&);          // (1.6)
private :
        int *ptab ;
        int nbe ;       // nombre d'éléments mémorisés dans la zone pointée par ptab
};
```

Développer entièrement cette classe, de façon à ce que la fonction f() ci-dessous s'exécute correctement :

```
void f()
{
        int tab[] = {13, -2, 17, -312, 177};
        Vecteur v1 (tab, (sizeof(tab) / sizeof(int)));   // 2ème paramètre : nombre d'éléments dans tab
        Vecteur v4 = v1 ;

        v4.set_val(3, 310) ;
        cout << "v4 = " << v4 << endl ;        // affiche        v4 = {13,  -2,  17,  310,  177}
}
```

2) Implémentation de vecteurs creux

Il s'agit maintenant de développer la classe `Vecteur_Creux` :

```
class Vecteur_Creux {
public :
        Vecteur_Creux (int n = 0);
        Vecteur_Creux (const Vecteur& le_vut, const Vecteur& le_vix, int n);
        Vecteur_Creux Sous_Vecteur (int d) const;
        // ...
private :
        Vecteur vut;
        Vecteur vix;
        int nbr;          // nombre d'éléments réels du vecteur creux
        int nbu;          // nombre d'éléments utiles du vecteur creux
};
```

2.1) Ecrire un premier constructeur de `Vecteur_Creux` de taille réelle n, où n est passé en argument du constructeur, de manière à allouer n éléments à `vut` et à `vix`, et à les initialiser à zéro.

2.2) Ecrire un second constructeur pour un `Vecteur_Creux` en lui passant en arguments son vecteur utile, son vecteur d'index, ainsi que la taille réelle du vecteur creux.

2.3) Ecrire la fonction membre `Sous_Vecteur` qui retourne une version réduite du vecteur creux courant : le nombre d'éléments réels est réduit à d (avec d < nbr), on ne garde donc que les indices utiles < d ; cette fonction sert au **3)**.

NB : seules les fonctionnalités qui viennent d'être demandées dans la classe `Vecteur_Creux` figurent dans (l'ébauche de déclaration ci-dessus de) la classe. Les fonctionnalités à venir sont entièrement à ajouter par vos soins. Les tester ensuite sur les vecteurs v2 et v3.

2.4) Est-il utile d'écrire un constructeur par copie et l'opérateur d'affectation pour cette classe? (justifier sous forme d'un commentaire).

2.5) Définir les fonctions membres ***taille_r()*** et ***taille_u()*** qui renvoient respectivement le nombre d'éléments réels et utiles d'un vecteur creux.

2.6) Surcharger l'opérateur << de manière à ce l'exécution de

```
cout << "[" << v3 << "]\n";
```

produise l'affichage suivant :

```
[  11    0    0 -612    0  403]
```

Il conviendra pour cela de formater les entiers sur 5 caractères …

2.7) Ecrire une fonction amie ***operator**** qui permet de calculer K * x, où x est un vecteur creux et K un scalaire entier.

2.8) Ecrire une fonction ***operator+*** d'addition de deux vecteurs creux. NB : dans l'absolu, le vecteur résultant peut comporter des éléments nuls ; il n'est pas demandé ici de les « supprimer ».

3) Gestion de matrices creuses

On cherche maintenant à définir et à manipuler des matrices creuses : des matrices dont de "nombreux" éléments sont nuls. Une telle matrice peut être représentée par une double liste : une liste de vecteurs creux, accompagnée d'une liste d'indices ***i***, chaque vecteur creux représentant une ligne ***i*** de la matrice.

Sur un exemple : soient

```
Vecteur_Creux tabVC[] = {v3, v2};
```

et

```
int tabInd[] = {0, 2};
```

Ces 2 "listes" (tableaux en réalité) permettent de spécifier entièrement la matrice creuse MC suivante, de dimension 3 x 7 :

```
        [  11    0    0  -612    0   403     0]
MC      [   0    0    0     0    0     0     0]
        [   0   -9    0    -5    0     0    28]
```

Une ébauche de structure pour la classe `Matrice_Creuse` et permettant de stocker de telles matrices creuses est fournie ci-dessous. On y mémorise également les dimensions de la matrice.

```
class Matrice_Creuse {
public :
    Matrice_Creuse (Vecteur_Creux* tVec, int* tInd, int nbut, int nbli);
    Matrice_Creuse (const Matrice_Creuse&);
    ~Matrice_Creuse ();
    Matrice_Creuse& operator= (const Matrice_Creuse&);
    // ...
protected :
    Vecteur_Creux* ligUt;       // lignes utiles
    int* indUt;                 // indices des lignes utiles
    int nbu;                    // nombre de lignes utiles
    int nbLig, nbCol;           // nb de lignes et de colonnes réelles
```

3.1) Définir entièrement les quatre fonctions membres déclarées dans la partie ***public***.

3.2) Surcharger dans `Matrice_Creuse` l'opérateur << afin d'afficher le contenu d'une matrice creuse.

3.3) Surcharger ***operator**** (multiplication d'une matrice creuse par un scalaire entier).

3.4) Surcharger ***operator+*** (ajout de 2 matrices creuses). Comme pour les vecteurs creux (cf. 2.8), on ne demande pas ici d'éliminer les éléments résultants nuls.

Tester entièrement cette classe.

4) Gestion de matrices carrées creuses

Une matrice carrée est une matrice de dimensions C x C. On s'intéresse ici bien évidemment à de telles matrices carrées … creuses.

Spécifier <u>entièrement</u> la classe `Matrice_Carree` héritant de la classe `Matrice_Creuse`. Le constructeur de `Matrice_Carree` devra recevoir en argument une `Matrice_Creuse` (de dimensions N x M) que l'on devra "réduire" de manière à ce que l'on ait C = min (N, M).

NB : on ne conservera de la matrice de base passée en paramètre que les C premières lignes et les C premières colonnes. Sur l'exemple de la matrice MC du 3), sa matrice carrée (dérivée) est :

```
        [ 11   0   0]
MCa     [  0   0   0]
        [  0  -9   0]
```

Comme précédemment (questions 2.8 et 3.3), on ne cherchera pas à "éliminer" les éléments nuls de la matrice carrée obtenue.

Par contre, il conviendra d'afficher le contenu d'une telle matrice carrée.