IFT339 - Introduction aux classes

Exercice 1

Implémentez une classe PointxD encapsulant un point multi-dimensionnel (le xD est pour x-dimensionnel), dans lequel le type des coordonnées et la dimension sont choisies par l'utilisateur. On doit pouvoir accéder aux coordonnées via l'opérateur [ ] et on doit pouvoir additionner deux points via + (utile pour faire des translations, par exemple). Un exemple d'utilisation:

```
PointxD<int, 4> p; //point 4 dimensions avec coordonnees int
//specifier les coordonnees
p[0] = 0;
p[1] = 1;
p[2] = 2;
p[3] = 3;
cout<<p[2]<<endl;

PointxD<int, 4> q;
q[1] = 10;
PointxD<int, 4> psomme = p + q;
cout<<psomme[1]<<endl; //affiche 11
```

Le type à stocker et la dimension doivent être templatées. Vous devez stocker les coordonnées dans un tableau de taille fixe, et donc démarrer avec

```
template <typename TYPE, unsigned int DIM>
class PointxD{
private:
    TYPE tab[DIM];
    //...
};
```

Les fonctionnalités à implémenter sont:
construction(): initialise toutes les coordonnées à 0.
operator[ ](i): retourne une référence vers l'élément à la i-ème coordonnée. L'opérateur doit permettre d'accéder et de modifier cette coordonnée. La signature habituelle est:

TYPE& operator[ ](size_t i)

dim(): retourne la dimension.
operator+(p): retourne un nouvel objet PointxD, où pour chaque $i = 0$ à DIM, la coordonnée i est la somme de la coordonnée i de l'objet courant et celle de p. La signature est:

PointxD operator+(PointxD& p)

Implémentez le destructeur, copieur, operator= seulement si nécessaire.

Solution

Voir fichiers sur la page du cours.

Fin de la solution

Exercice 2

Implémentez une classe encapsulant un tableau 1D de taille dynamique qui stocke des éléments de type arbitraire. Vous aurez besoin des pointeurs. Un exemple d'utilisation:

```
Tableau<double> tab(10);
tab[2] = 1.1;
cout<<tab[2]<<endl;
tab.redimensionner(5);
cout<<tab.size()<<endl;
```

Votre implémentation doit stocker le tableau avec une variable membre TYPE* tab;, dont la taille est dynamique. Les fonctionnalités à implémenter sont:
construction(n): créé un tableau de taille n (avec new).
destructeur(): libère la mémoire du tableau.

copieur(src): constructeur par copie.
operator[](i): retourne une référence (non-const) vers le i-ème élément.
redimensionner(n): met la taille à n éléments, sans supprimer les éléments encore présents.
size(): retourne n.
operator=(src): affectateur.

Solution

Voir fichiers sur la page du cours.

Fin de la solution

Exercice 3

Donnez le code C++ permettant de créer un tableau d'entiers en 3 dimensions dynamique, à l'aide d'un int***. Vous devez affecter 0 à chaque cellule et supprimer toute la mémoire allouée.

Solution

```
//dimensions n, m, p
int*** tab = new int[n];
for (int i = 0; i < n; ++i){
    tab[i] = new int[m];

    for (int j = 0; j < m; ++j){
        tab[i][j] = new int[p];

        for (int k = 0; k < p; ++k){
            tab[i][j][k] = 0;
        }
    }
```

```
}

//suppressions
for (int i = 0; i < n; ++i){
    for (int j = 0; j < m; ++j){
        delete [] tab[i][j];
    }
    delete [] tab[i];
}
delete [] tab;
```

Fin de la solution

Exercice 4

Supposons que vous avez deux tableaux dynamiques en C++ déclarés comme suit:

```
int* tab1 = new int[n];
int* tab2 = new int[m];
```

où n et m sont des entiers quelconques. Donnez le code qui échange le contenu de $tab1$ avec celui de $tab2$ (donc après exécution, le contenu de $tab1$ est devenu celui de $tab2$, et celui de $tab2$ est devenu celui de $tab1$).

Solution

```
int* tmp = tab1;
tab1 = tab2;
tab2 = tmp;
```

Fin de la solution

Exercice 5

Condidérez la classe BitVec ci-dessous. Le code compile correctement, mais il y a toutefois beaucoup de problèmes avec cette classe par rapport aux principes vus en classe. Identifiez au moins trois problèmes avec la conception de cette classe, et indiquez comment les corriger (note: ne dites pas "il n'y a pas de commentaire", ce n'est pas la réponse recherchée).

```
class BitVec
{
public:
    vector<bool>* v;

    BitVec()
    {
        v = new vector<bool>(10);
        for (size_t i = 0; i < v->size(); i++)
            (*v)[i] = true;
    }

    BitVec(const BitVec& src)
    {
        this->v = src.v;
    }

    BitVec* getReverse()
    {
        BitVec* copie = new BitVec();
        for (size_t i = 0; i < v->size(); i++)
        {
            (*copie->v)[i] = !(*v)[i];
        }
        return copie;
    }
};
```

Solution

1) Le vecteur de stockage v est public. Ceci brise le principe d'encapsulation, qui dit que la représentation interne d'une classe ne doit pas être accessible.

2) La variable v est créée avec un new, mais il n'y a pas de destructeur. Il y a donc fuite de mémoire.

3) Le copieur (constructeur par copie) copies le pointeur src.v, mais ceci ne copie pas la structure de src. Donc, modifier this va aussi modifier src.

4) getReverse fait un new et retourne le résultat. L'appelant n'a aucune façon de savoir qu'il devra faire un delete.

Fin de la solution