Algorithmique et Programmation C++

TD 6

1. Algorithmes et structures élémentaires

```
/*                       */
/*   TD 6 - Exercice 1   */
/*    Jeremie::Osmont    */
/*                       */


#include<iostream.h>
#include<stdlib.h>
#include<iomanip.h>


// Constantes :
const int M=4;
const int N=30;


// Creation de la matrice qui contiendra les morpions :
int A[M+2][N+2];


// Generateur de nombres aleatoires :
long rnd(long max)
{
    long a=rand();
    while (a>=10000)
        a = a - 10000;
    return static_cast<long>(max*a/10000);
}


// Fonction qui calcule le nombre de voisins pour un
// morpion d'abscisse x et d'ordonnee y :
int voisins(int G[M+2][N+2], int x, int y)
{
    // Cette variable contiendra le nombre de voisins :
    int res=0;

    // On fait le tour de la case (x,y) avec deux boucles :
    for(int i=x-1; i<=x+1; i++)
        for(int j=y-1; j<=y+1; j++)
            // Ce test permet de ne pas compter la case
            // de coordonnees (x,y) !
            if(!(i==x && j==y))
                res+=G[i][j];

    // Autre methode (non demandee dans l'enonce),
    // qui considere que la matrice est 'circulaire' :
    // chaque bord est adjacent au bord oppose (un
    // peu comme les tableaux de Karnaugh...).
    /*
    res+=G[(x+1)%M][(y+1)%N];
    res+=G[(x+1)%M][y];
    res+=G[(x+1)%M][(y-1+N)%N];
    res+=G[x][(y+1)%N];
    res+=G[x][(y-1+N)%N];
    res+=G[(x-1+M)%M][(y+1)%N];
    res+=G[(x-1+M)%M][y];
    res+=G[(x-1+M)%M][(y-1+N)%N];
    */
    // En effet, cette methode n'a pas besoin d'un
    // systeme de 'centrage' dans une matrice +2...
    return res;
}


// Fonction qui calcule une generation :
void gensuiv(int G[M+2][N+2])
{
    // Pour une generation, on travaille avec une autre
    // matrice S, de facon a ne pas modifier en cours
    // de generation la matrice G (pour eviter les
    // fameux effets de bord).
    int S[M+2][N+2];
    int nvoisins;
    // On parcours toutes les cases de la matrice G a l'aide
    // de deux boucles for imbriquees :
    for(int i=1; i<M+1; i++)
        for(int j=1; j<N+1; j++)
        {
            // Recuperation du nombre de voisins pour une case
            // determinee :
            nvoisins=voisins(G,i,j);
            // Puis application des modifications dans S :
            if(nvoisins<2) S[i][j]=0;// Encore un morpion sans ami...
            if(nvoisins==2) S[i][j]=G[i][j];// Survivant
            if(nvoisins==3) S[i][j]=1;// Un nouveau-ne !
            if(nvoisins>3) S[i][j]=0;// Mort etouffe...
        }

    // Recopie de S dans G (il suffit de copier l'interieur,
    // car la bordure est constituee de 0) :
    for(int a=1; a<M+1; a++)
        for(int b=1; b<N+1; b++)
            G[a][b]=S[a][b];
}


// Programme principal de lancement :
int main()
{
    // Astuce : pour ne pas avoir a gerer le cas des bords,
    // la matrice A sera de dimensions +2 par rapport a M et N.
    // Cela permet de centrer la colonie de morpions dans une
    // matrice pleine de 0, comme cela (pour M=N=4) :
    // 0 0 0 0 0 0
    // 0 1 0 1 1 0
    // 0 0 1 1 1 0
    // 0 1 0 0 1 0
    // 0 1 1 1 0 0
    // 0 0 0 0 0 0

    // Demande le nombre de generations :
    int n;
    cout << "\nCombien de generations de petits morpions voulez-vous
generer ?\n";
    cin >> n;

    // Remplissage de zeros de A :
    for(int a=0; a<M+2; a++)
        for(int b=0; b<N+2; b++)
            A[a][b]=0;

    // Remplissage aleatoire de A :
    for(int k=1; k<M+1; k++)
        for(int l=1; l<N+1; l++)
            A[k][l]=rnd(2);

    // Calcule n generations :
    for(int i=0; i<n; i++)
    {
        cout << "\n* Generation #" << i << " :\n";
        // Affichage de A (n'affiche que la partie utile) :
        for(int a=1; a<M+1; a++)
        {
            for(int b=1; b<N+1; b++)
                cout << setw(2) << A[a][b];
            cout << endl;
        }
        // Calcul de la generation suivante :
        gensuiv(A);
    }
    return 0;
}
```

2. Récursivité

```
/*                       */
/*   TD 6 - Exercice 2   */
/*    Jeremie::Osmont    */
/*                       */

#include<iostream.h>

// Cette fonction recursive deplace n disques de la tige x
// a la tige z en utilisant la tige y :
void deplacer(int n, char x, char y, char z)
{
    if(n!=1)
    {
        // Cas general (hypothese de recurrence)
        deplacer(n-1,x,z,y);
        deplacer(1,x,y,z);
        deplacer(n-1,y,x,z);
    }
    else
        // Cas de base (pour n=1):
        cout << x << "->" << z << " ;  ";
}

// Programme de lancement :
int main()
{
    cout << "\nCombien de disques voulez vous deplacer ?\n";
    int i;
    cin >> i;
    cout << "\nDeplacements successifs :\n";
    deplacer(i, 'A', 'B', 'C');
    cout << endl;
    return 0;
}
```

3. Types complexes

```
/*                      */
/*   TD 6 - Exercice 3  */
/*    Jeremie::Osmont   */
/*                      */


#include<iostream.h>
#include<iomanip.h>


// Definition de la classe Pile :
class Pile
{
private:
    int max;
    int nbre;
    int * tab;

public:

    // Constructeur
    Pile(int n=20)
    {
        max=n;
        nbre=0;
        tab=new int[n];
        cout << "Creation d'une pile." << endl;
    }

    // Constructeur par recopie
    Pile(const Pile & p)
    {
        max=p.max;
        nbre=p.nbre;
        tab=new int[p.max];
        for(int i=0; i<nbre; i++)
            tab[i]=p.tab[i];
        cout << "Recopie d'une pile." << endl;
    }

    // Destructeur
    ~Pile()
    {
        delete tab;
        cout << "Destruction d'une pile." << endl;
    }

    // Ajoute un entier
    void empile(const int & n)
    {
        if(nbre<max)
            tab[nbre++]=n;
    }

    // Enleve un entier
    int depile()
    {
        if(nbre>0)
            return tab[--nbre];
        else
            return 0;
    }

    // La pile est-elle pleine ?
    bool pleine()
    {
        return nbre==max;
    }


    // La pile est-elle vide ?
    bool vide()
    {
        return nbre==0;
    }
};


// Programme de test de la classe :
int main()
{
    // Cree une pile vide :
    Pile duracell;
    // Cree une autre pile vide plus grande :
    Pile energizer(50);
    // Remplit une pile :
    int i=0;
    while(!duracell.pleine())
    {
        duracell.empile(i);
        cout << "Empilage de " << setw(2) << i << ", ";
        i++;
    }
    cout << "Pile pleine !\n";
    // Cree encore une autre pile a partir d'une existante :
    Pile philips(duracell);
    // Vide cette pile :
    int r;
    while(!philips.vide())
    {
        r=philips.depile();
        cout << "Depilage de " << setw(2) << r << ", ";
    }
    cout << "Pile vide !\n";
    return 0;
}
```

4. Listes chaînées

```
/*                      */
/*   TD 6 - Exercice 4  */
/*    Jeremie::Osmont   */
/*                      */


#include<iostream.h>
#include<iomanip.h>


// Maillon de base :
struct Maillon
{
    int info;
    Maillon * suiv;
};


// Definition de la classe Liste :
class Liste
{
public:
    Maillon * debut;
public:
    // Pour l'explication de ces fonctions, voir le TD5
    Liste() {debut=NULL;}
    ~Liste();
    void inserer(int);
    bool supprimer(int);
    void affiche();
    // Fonctions demandees :
    void inverser();
    void trier();
};


Liste::~Liste()
{
    Maillon * p;
    Maillon * courant=debut;
    while(courant!=NULL)
    {
        p=courant;
        courant=courant->suiv;
        delete p;
    }
}


void Liste::inserer(int x)
{
    Maillon * m=new Maillon;
    m->info=x;
    m->suiv=debut;
    debut=m;
}


bool Liste::supprimer(int x)
{
    Maillon * m=debut;
    Maillon * tmp;
    if (debut->info==x)
    {
        debut=m->suiv;
        delete m;
        return true;
    }
    while ((m->suiv)&&(m->suiv->info!=x)) m=m->suiv;
    if (m->suiv->info==x)
    {
        tmp=m->suiv;
        m->suiv=m->suiv->suiv;
        delete tmp;
        return true;
    }
    return false;
}


void Liste::affiche()
{
    Maillon *m=debut;
    while(m!=NULL)
    {
        cout << m->info << ";";
        m=m->suiv;
    }
    cout << endl;
}
```

```
// Inversion d'une Liste (sans la recopier) :
void Liste::inverser()
{
    // Cas de la liste nulle :
    if(debut==NULL) return;
    // Cas general :
    Maillon * r, * s;
    r=debut->suiv;
    // Le premier maillon sera le dernier :
    debut->suiv=NULL;
    // Puis parcourt toutes la Liste, en connectant a chaque
    // fois le maillon en cours au precedent (c'est pour ca qu'il
    // faut deux pointeurs : r stocke le maillon en cours et s
    // le suivant.) :
    while(r!=NULL)
    {
        s=r->suiv;
        r->suiv=debut;
        debut=r;
        r=s;
    }
}


// Tri selectif : on recherche le plus petit element
// et on le place devant, et on continue jusqu'a la fin :
void Liste::trier()
{
    // Cas de la liste nulle :
    if(debut==NULL) return;
    // Cas general - Explications (cette methode est peut etre pas la
    // plus optimisee, mais elle marche et au moins ca entraine sur
    // les pointeurs de maillons...
    // Donc on va travailler sur deux sous listes : une liste qui
    // sera non triee, et une liste triee. Le but du jeu sera de
    // connecter au fur et a mesure les maillons de la premiere
    // liste vers la seconde - dans l'ordre, justement.

    // Ce pointeur de maillon sert a parcourir la liste non-triee :
    Maillon * m;
    // Ce pointeur pointe vers le maillon avant m :
    Maillon * prec;
    // Ce pointeur pointe vers l'element le plus petit :
    Maillon * mini;
    // Ce pointeur pointe vers l'element avant le plus petit :
    Maillon * miniprec;
    // Ce pointeur designe le premier maillon de la liste non-triee :
    Maillon * depart=debut;
    // Ce pointeur designe le dernier maillon de la liste TRIEE
    // (qui est vide au debut) :
    Maillon * t=NULL;

    // Tant que la liste non-triee n'est pas vide...
    while(depart!=NULL)
    {
        // On positionne m au debut de la liste non-triee :
        m=depart;
        prec=NULL;
        // Par hypothese le plus petit est le premier :
        mini=depart;
        miniprec=NULL;

        // Recherche le plus petit maillon :
        while(m!=NULL)
        {
            if(m->info < mini->info)
            {
                // On a trouve un minimum, on l'enregistre dans mini :
                mini=m;
                miniprec=prec;
            }
            // De cette maniere, prec est toujours le maillon devant m
            prec=m;
            m=m->suiv;
        }

        // Ici on a trouve le plus petit des non tries, c'est mini :
        if(mini==depart)
        {
            // Cas particulier - le plus petit est le premier
            // des non-tries :
            if(t==NULL)
            {
                // Sous cas particulier, la liste des tries est vide.
                // Donc notre plus petit element sera au debut de la
                // liste totalement triee :
                debut=depart;
                // t designe le dernier des maillons tries :
                t=debut;
            }
            else
            {
                // Sous cas general, notre plus petit element est
                // au milieu de la liste :
                t->suiv=depart;
                t=t->suiv;
            }
            // Dans les deux cas, depart ne fait plus partie de la
            // Liste des non tries, donc on le fait passer au
            // maillon d'apres :
            depart=depart->suiv;
        }
        else
        {
            // Cas general : le plus petit est au milieu, il faut
            // donc l'extraire et le connecter a la fin de la
            // sous liste triee :
            if(t==NULL)
            {
                // Sous cas particulier, la liste des tries est vide :
                debut=mini;
                t=debut;
            }
            else
            {
                // Sous cas general :
                t->suiv=mini;
                t=t->suiv;
            }
            // Dans les deux cas, ceci sert a 'sauter' l'element mini
            // qui a ete retire de la sous liste non triee :
            miniprec->suiv=mini->suiv;
        }
    }
    // Pour faire plus simple : au lieu de modifier les pointeurs, de
    // s'amuser a deconnecter des maillons et a les changer de place,
    // on aurait pu simplement changer les info des maillons, en fait
    // ca reviendrait a traiter la Liste comme un tableau.
}


// Programme de test :
int main()
{
    Liste l;
    int ent=1;
    cout << "\nEntrez des entiers pour remplir la liste (0 pour
arreter) :\n";
    while(ent!=0)
    {
        cin >> ent;
        if(ent!=0) l.inserer(ent);
    }
    cout << "Liste entree :\n";
    l.affiche();
    cout << "Liste inversee :\n";
    l.inverser();
    l.affiche();
    cout << "Liste triee dans l'ordre croissant :\n";
    l.trier();
    l.affiche();
    cout << "Liste triee dans l'ordre decroissant :\n";
    l.inverser();
    l.affiche();
    return 0;
}
```