Université Louis Pasteur
U.F.R. de Mathématiques
Strasbourg 1

Licence de Mathématiques
Année 2002/2003

EXAMEN d'INFORMATIQUE

Session 1 – 27 janvier 2003

CORRIGE

Dans tout le sujet, on considère le plan euclidien, muni d'un repère (O, $\vec{i}$, $\vec{j}$), de la distance issue de la norme euclidienne ($||\vec{u}|| = \sqrt{x_u^2 + y_u^2}$) et de l'ordre lexicographique sur les points. On rappelle que l'ordre lexicographique consiste à comparer d'abord les abscisses, puis en cas d'égalité les ordonnées.
Le sujet comporte 3 pages. Les parties ne sont pas indépendantes. Le barème est donné à titre indicatif.

Partie I : Points et droites *(6 points)*

1. On implante de manière basique la notion de point *nommé* du plan à l'aide de la classe `point` déclarée de la manière suivante (on rappelle que la fonction `strcpy` copie la chaîne de caractère donnée en deuxième argument dans la chaîne de caractères donnée en premier argument, et que la fonction `strcat` copie la chaîne en deuxième argument *à la suite* de celle donnée en premier argument).

```
// fichier PointDroite.h
#include <iostream>
#include <string.h>
class point
{    char  n[32];     // nom du point
     float absc;      // abscisse
     float ord;       // ordonnée

    public:
           point()  { absc = 0; ord = 0; strcpy(n,"O"); }
           point(float x, float y, char *nom_point);
     void  get();
     float X();                  // retourne l'abscisse
     float Y();                  // retourne l'ordonnée
     char* nom();                // nom du point courant
     float dist(point P);        // distance du point P au point courant
     bool  operator==(point P);  // test d'égalité (sans tenir compte du nom)
     bool  operator <(point P);  // ordre lexicographique
    friend ostream& operator<<(ostream& o, point P);
}; // fin de la classe point
```

Donnez le code de toutes les fonctions membres de la classe `point`.

```
// ********************** Déclaration des constructeurs **********************
point::point(float x, float y, char *nom_point)
{   absc = (x-float(int(x))<1e-5 && x+float(int(x))>-1e-5 ? float(int(x)) : x);
    ord  = (y-float(int(y))<1e-5 && y+float(int(y))>-1e-5 ? float(int(y)) : y);
    strcpy(n, nom_point);
} // ces lignes remplacent -2.54e-08 par 0 (erreurs d'arrondi...)

// ********************* Déclaration des fonctions amies *********************
ostream& operator<<(ostream& o, point P)
{   o << P.nom() << "(" << P.X() << " ; " << P.Y() << ")";
    return o;
} // on affiche un point selon l'exemple suivant : A(1 ; 2.5)

// ******************** Déclaration des fonctions membres ********************
```

```
void point::get()
{   char NOM[32];
    cout << "Saisie d'un point : " << endl;
    cout << "* Nom du point : "; cin >> n;
    cout << "* Abscisse : ";     cin >> absc;
    cout << "* Ordonnée : ";     cin >> ord;
}

float point::X()     { return absc; }
float point::Y()     { return ord;  }
char* point::nom()   { return n;    }

float point::dist(point P)
{   return sqrt((absc-P.absc)*(absc-P.absc) + (ord-P.ord)*(ord-P.ord));
}

bool point::operator==(point P)
{   return (fabs(P.abs-abs)<1e-06 && fabs(P.ord-ord)<1e-06);
}

bool point::operator<(point P)
{   return (absc==P.absc ? ord<P.ord : absc<P.absc);
}
```

2. On propose la classe `droite` modélisant la notion classique de droite du plan :

```
class droite
{    ... // à définir
    public:
           droite() ;
           droite(point A, point B) ;
           // ... éventuellement d'autres constructeurs
     void  get();                 // Saisie
     float coef();                // coefficient directeur
     float ord();                 // ordonnée à l'origine
     char* nom();                 // nom de la droite
     bool  vert();                // test de verticalité
     bool  horiz();               // test d'horizontalité
     bool  operator==(droite D);  // test d'égalité
     bool  contient(point P);     // test d'incidence
    friend ostream& operator<<(ostream& o, droite D);
}; // fin de la classe droite
```

Précisez les préconditions inhérentes à chacune des fonctions de la classe `droite`. Proposez une structure de données pour la classe `droite` et programmez toutes les fonctions membres indiquées.

```
// suite du fichier PointDroite.h
class droite
{    char  N[32];
     float CoeffDir;  bool V;  // true si la droite courante est verticale
     float OrdOrig;

    public:
           droite() { CoeffDir=0; OrdOrig=0; V=false; strcpy(N, "0x"); }
           droite(point A, point B);
           droite(float CD, float OO, char *nom_droite);
     void  get();
     float coef();
     float ord();
     char* nom()    { return N; }
     bool  vert()   { return V; }
```

```
     bool  horiz()  { return (CoeffDir==0); }
     bool operator==(droite D);
     bool contient(point P);
    friend ostream& operator<<(ostream& o, droite D);
};


// ********************** Déclaration des constructeurs **********************
droite::droite(point A, point B)
{   // on présuppose que les deux points sont distincts
    V = (B.X() == A.X());
    CoeffDir = (V ? 0 : (B.Y()-A.Y())/(B.X()-A.X()));
    // La ligne ci-dessus sert juste à éviter l'arrêt du programme si A.X()=B.X()
    OrdOrig  = (V ? A.X() : A.Y() - CoeffDir*A.X());
    strcpy(N, strcat(A.nom(), B.nom()));
}

droite::droite(float CD, float OO, char *nom_droite)
{   CoeffDir = CD;
    OrdOrig  = OO;
    strcpy(N, nom_droite);
    V = true;
}

// ********************* Déclaration des fonctions amies *********************
ostream& operator<<(ostream& o, droite D)
{   if(D.vert())  o << D.nom() << " : x = " << D.ord();
    else          o << D.nom() << " : y = " << D.coef() << "x + " << D.ord();
    return o;
} // on affiche un point selon l'exemple suivant : (AB) : y = 2x + 2.5

// ******************** Déclaration des fonctions membres ********************
void droite::get()
{   char NOM[32]; int choix = 1; point A, B;
    cout << "Choix de la saisie d'une droite : " << endl;
    cout << "1. par son équation de droite" << endl;
    cout << "2. par deux points la définissant" << endl;
    cout << "Votre choix : "; cin >> choix;
    choix = (choix<0 || choix>2 ? 1 : choix);

    if (choix == 1)
    {   cout << "Saisie d'une droite : " << endl;
        cout << "* Nom de la droite : "; cin >> NOM;
        cout << "* Coefficient directeur : ";     cin >> CoeffDir;
        cout << "* Ordonnée à l'origine  : ";     cin >> OrdOrig;
        strcpy(N, NOM); V = false;
    }
    else
    {   cout << "Saisie d'une droite par deux points : " << endl;
        A.get(); B.get();
        *this = droite(A, B);
    }
}

float droite::coef()
{   // On présuppose que la droite courant n'est pas verticale
    if (V)   { cerr << "droite.coef:droite verticale"; exit(1); }
    else     return CoeffDir;
}

float droite::ord()
{   if (V)   cerr << "droite.ord:droite verticale";
    return OrdOrig;  // Par définition, si la droite courante est verticale, alors
                     // on renvoie son « abscisse », sinon l'ordonnée à l'origine.
}
```

```
bool droite::operator==(droite D)
{   if (!V && !D.V)
        return (fabs(CoeffDir-D.coef())<1e-06 && fabs(OrdOrig-D.ord())<1e-06);
    else if (V && D.V) return (fabs(OrdOrig-D.OrdOrig)<1e-06);
    else               return false;
}

bool droite::contient(point P)
{   return (V ? OrdOrig==P.X() : P.Y() == CoeffDir*P.X() + OrdOrig);
}
```

3. Modifiez la classe `point` donnée plus haut en ajoutant un constructeur de point défini par deux droites. Programmez une fonction C++ de profil : **`bool`** `alignes(point A, point B, point C);` testant l'alignement de trois points.

On ajoute la ligne ci-dessous dans la classe `point` :
`point(droite D1, droite D2);`
On aura pris soin de rajouter la ligne « **`class`** `droite;` » juste avant la ligne **`class`** `point`, ceci est nécessaire pour le troisième constructeur qu'on définit à l'aide de `droite`, dont la classe n'est elle-même définie qu'ultérieurement. Une pré-déclaration est donc nécessaire, ce qui est l'objet de cette ligne.

On la définit hors de la classe `point` en écrivant :

```
point::point(droite D1, droite D2)
{   if (D1.vert())
    {   if (d2.vert())   { cerr << "droites parallèles"; exit(1); }
        absc = D1.ord(); ord = D2.coef()*absc+D2.ord();
    }
    else
    {   if (D2.vert())   { absc = D2.ord(); ord=D1.coef()*absc+D1.ord(); }
        else
        {   if(fabs(D1.coef()-D2.coef()<1e-06)
            {   cerr << "Droites parallèles";
                exit(1);
            }
            absc = (D2.ord()-D1.ord())/(D1.coef()-D2.coef());
            ord  = D1.coef()*abs + D1.ord();
        }
    }
    strcpy(n, "I("); // comme intersection
    strcat(n, D1.nom()); stracat(n, ","); strcat(n,D2.nom()); strcat(n,")");
}
```

Alignement de trois points (2 versions possibles) :

```
bool alignes(point A, point B, point C)
{   droite D1(A,B), D2(A,C);
    return (D1 == D2);
}
```

```
bool alignes(point A, point B, point C)
{   droite D1(A,B);
    return (D1.contient(C));
}
```

Partie II : Liste de points, liste de droites *(6 points)*

On considère dans cette section des listes de points et des listes de droites avec les opérations classiques d'insertion dans une liste, d'accès au ième élément d'une liste, etc.

```
// fichier listpoint.h
#include <iostream>

class listpoint
{    // à préciser
```

```
    public:
           listpoint();                    // liste vide
     void  a_ieme(point A, int pos);       // ajouter en pos-ième position
     void  r_ieme(int pos);                // retirer en pos-ième position
     point operator[](int i);              // accès au ième élément
     int   longueur();                     // longueur
     bool  est_vide();                     // test de vacuité
     bool  contient(point P);              // test d'appartenance
    friend ostream& operator<<(ostream& o, listpoint l);
}; // fin de la classe listpoint
```

1. On suppose qu'on n'a jamais plus de 128 points dans une liste et on choisit une implantation contiguë. Proposer une structure de donnée pour la classe `listpoint` et programmez les fonctions membres.
Sur le modèle précédent, proposez une définition de la classe correspondant à la notion de liste de droites (vous donnerez une structure de données et les prototypes des fonctions membres. *Sans détailler leur implémentation*, indiquez comment on peut les obtenir à partir des fonctions membres de la classe `listpoint`.

```
class listpoint
{    point POINTS[128];
     int dim_P;
    public:
           listpoint()  { dim_P = 0; }
     void  a_ieme(point A, int pos);
     void  r_ieme(int pos);
     point operator[](int i);
     int   longueur()         { return dim_P; }
     bool  est_vide()         { return (dim_P == 0); }
     bool  contient(point P);
    friend ostream& operator<<(ostream& o, listpoint LP);
}; // fin de la classe listpoint


// ******************** Déclaration des fonction amies **********************
ostream& operator<<(ostream& o, listpoint LP)
{   cout << "{ ";
    for (int i=0; i<LP.longueur()-1; i++)
        cout << LP[i] << " ; ";
    cout << LP[LP.longueur()-1] << " }";
    return o;
}

// ******************** Déclaration de fonctions membres ********************
void listpoint::a_ieme(point A, int pos)
{   int i=dim_P;
    for (; i>=pos; i--)
        POINTS[i] = POINTS[i-1];
    POINTS[i-1] = A;
    dim_P++;
}

void listpoint::r_ieme(int pos)
{   for (int i=pos-1; i<dim_P-1; i++)
        POINTS[i] = POINTS[i+1];
    dim_P--;
}

point listpoint::operator[](int pos)
{   if (pos>nb || pos<1)
    {   cerr << "listpoint.[] : Hors de la liste"; exit(1);
    }
    else return POINTS[i-1];
}

bool listpoint::contient(point P)
```

```
{   for (int i=0; i<dim_P; i++)
        if (P == POINTS[i])  return true;
    return false;
}
```

Structure de données de la classe `listdroite` et prototypes des fonctions membres :

```
class listdroite
{    droite DROITES[127];
     int dim_D;
    public:
            listdroite();
     void   a_ieme(droite D, int pos);
     void   r_ieme(int pos);
     droite operator[](int i);
     int    longueur();
     bool   est_vide();
     bool   contient(droite D);
    friend  ostream& operator<<(ostream& o, listdroite LD);
}; // fin de la classe listdroite
```

On remplace les lignes de la classe `point` par celles-ci (on n'oubliera pas d'apporter les modifications `point` → `objet` dans la définition de chaque fonction, après la classe) :

```
typedef point objet;
class liste
{    objet POINTS[127];
     int dim_P;
    public:
           liste()  { dim_P = 0; }
     void  a_ieme(objet A, int pos);
     void  r_ieme(int pos);
     objet operator[](int i);
     int   longueur()         { return dim_P; }
     bool  est_vide()         { return (dim_P == 0); }
     bool  contient(objet P);
    friend ostream& operator<<(ostream& o, liste LP);
}; // fin de la classe liste
typedef liste listpoint;
```

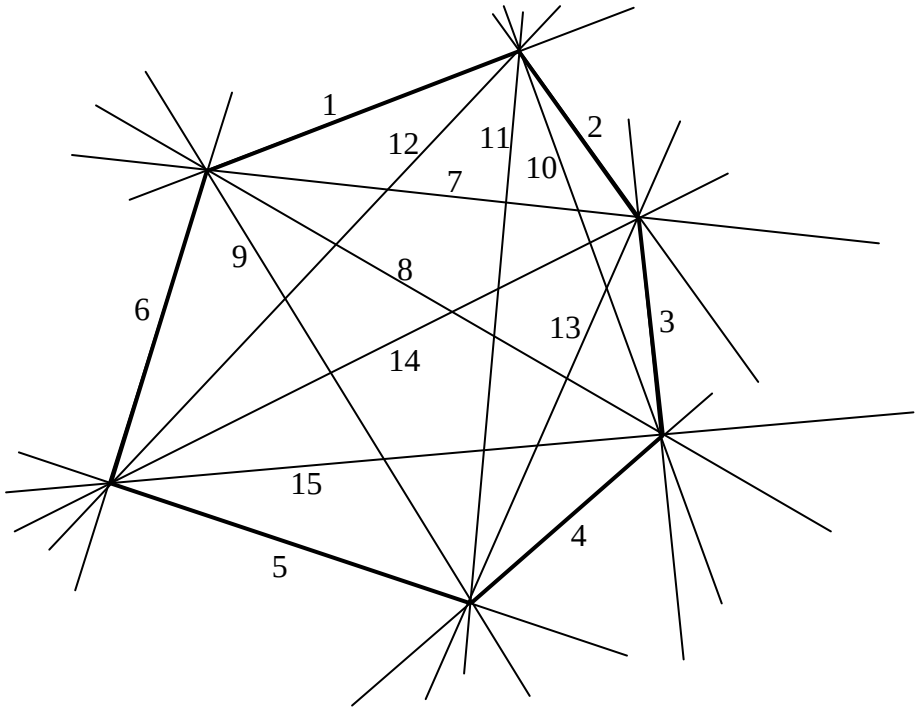
La première ligne signifie que `objet` sera un synonyme de `point`. La dernière signifie alors que `listpoint` sera synonyme de `liste`. En remplaçant alors `point` par `droite` dans la première ligne, et `listpoint` par `listdroite` dans la dernière, on aura "créé" la classe `listdroite` à partir des fonctions de celle de `listpoint`. Le seul inconvénient étant qu'on ne peut pas utiliser simultanément les classes `listpoint` et `listdroite` par ce procédé.

2. Programmez une fonction de prototype `listdroite toutes_les_droites(listepoint E);` produisant à partir d'une liste de points la liste, sans omission et *sans redondance*, de toutes les droites qu'on peut définir par alignement à partir de ces points. Combien y en a-t-il au plus ?
Programmez une fonction de prototype **`void`** `tous_les_alignes(listepoint E);` *affichant par paquets* les sous-listes de points alignés de E.

```
listdroite toutes_les_droites(listpoint E)
{   listdroite D; droite d;
    for (int i=0; i<E.longueur()-1; i++)
        for (int j=i+1; j<E.longueur(); j++)
        if (!(E[i]==E[j]))
        {   d = droite(E[i], E[j]);
            if (!D.contient(d))    D.a_ieme(d, D.longueur()+1);
        }
```

```
    return D;
}
```

Par exemple, dans le cas d'un hexagone défini par $n = 6$ points, le calcul donne alors qu'au maximum, 15 droites sont définissables à partir de ces 6 points, comme le montre le schéma ci-dessous :



Quelques explications de la fonction s'imposent : le premier for sert à partir tous les points jusqu'à l'avant-dernier (on s'arrête à l'avant dernier car on va créer des droites avec ce point et tous ceux qui suivent, donc il en faut au minimum un !). Le deuxième for servira à construire les droites avec le point du premier for. A chaque *j*, on construit la droite qui correspond au point courant avec le point du *i*, on l'ajoute à la liste de droite. Le 3ème for servira à comparer la droite courante à toutes celle déjà crées. S'il existe une droite identique, on supprime la droite courante de la liste, sinon, on la laisse. Cet algorithme nous permet facilement de calculer qu'il peut y avoir au maximum $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ droites, où n = `E.longueur()` est le nombre de points de la `listpoint` E.

Fonction `tous_les_alignes` :

```
void tous_les_alignes(listpoint E)
{   bool virgule = false;
    cout << " { ";
    for (int i=0; i<E.longueur()-2; i++)
        for (int j=i+1; j<E.longueur()-1; j++)
            for (int k=j+1; k<E.longueur(); k++)
            {   if (alignes(E[i], E[j], E[k]))
                {   if (virgule) cout << " , ";
                    cout << "<" << E[i] <<" ; "<< E[j] <<" ; "<< E[k] <<">"<< endl;
                }
                virgule = true;
            }
    cout << " }";
    return;
}
```

Le principe est le même que tout à l'heure sauf que pour *i*, on s'arrête à l'avant-avant-dernier point car il

faut bien le comparer à au moins deux points suivants.

3. Programmez une fonction de prototype `listepoint completePoint(listpoint E);` produisant la liste, sans omission et sans redondance, des points obtenus à partie de E par intersection de toutes les droites pouvant être définies dans E. Combien y a-t-il de points au plus dans cette nouvelle liste ?

```
listpoint completePoint(listpoint E)
{   listdroite d; listpoint p;
    d = toutes_les_droites(E);
    for (int i=0; i<d.longueur()-1; i++)
        for (int j=i+1; j<d.longueur(); j++)
        {   if (fabs(d[i].coef()-d[j].coef())<1e-06) break;    // (*)
            p.a_ieme(point(d[i], d[j]), p.longueur());
            for (int k=0; k<p.longueur()-1; k++)
                if (p[p.longueur()-1] == p[k])   p.r_ieme(p.longueur()-1);
        }
    return p;
}
```

On remarquera que la ligne `(*)` sert à éliminer le cas où `d[i]` et `d[j]` seraient parallèles, auquel cas il n'y aurait pas de points d'intersection…
Pour un ensemble de départ de *n* points distincts (où *n* représente encore `E.longueur()`), on ajoute au plus $n(n-1)(n-2)(n-3)/8$ points « nouveaux ». En effet, quatre points disctincts définissent deux droites qui se coupent en un point autre qu'un point déjà présent dans la liste, ce qui explique qu'on prenne $n(n-1)(n-2)(n-3)$. Mais on peut faire des permutations de points pour définir une droite et des permutations de droites pour définir un point. Il y en a respectivement 4 et 2, ce qui fait qu'on divise le nombre précédent par 4 * 2 = 8. Comme ce sont des points nouveaux qui sont créés, il suffit alors d'ajouter les *n* points initiaux, d'où ce résultat final : $\frac{n(n-1)(n-2)(n-3)}{8} + n$.

Partie III : Hexagones, hexamys et polygones *(8 points)*

On considère dans cette partie des hexagones, ou plus précisément des sextuplés de *points distincts*. L'implantation sera faite par la classe `hexagone` ayant pour structure de données un tableau de 6 points.

1. Donnez une première version de la classe `hexagone` précisant les structures de données, les constructeurs et quelques fonctions que vous jugerez utiles (affichage, comparaison, modification, …). Programmez les constructeurs que vous avez prévus.

```
// fichier hexamy.h
class hexagone
{    point h[6];
     char NH[32];
    public:
           hexagone();
           hexagone(point centre);  // hexagone régulier centré (exercice)
           hexagone(point A, point B, point C, point D, point E, point F);
     char* nom();
     point operator[](int i);              // accès au (i-1)-ème point de l'hexagone
     void  modification(point P, int pos);  // modification (avec vérification)
     void  get();                           // saisie d'un hexagone
     bool  est_un_hexamy();                 // teste si un hexagone est un hexamy
     bool  operator==(hexagone H2);         // test d'équivalence
    friend ostream& operator<<(ostream& o, hexagone H);
}; // fin de la classe hexagone
```

La fonction **operator**==(hexagone H2) renverra true si l'hexagone H2 est égal à l'hexagone courant. On dit ici que deux hexagones sont égaux si leurs premiers points sont égaux dans la liste, leurs deuxièmes aussi, etc.
Les fonctions void get() et char* nom() et sont les fonctions usuelles de saisie et de renvoi de nom de l'hexagone. La fonction amie est la fonction standard d'affichage d'un hexagone.

Programmation des constructeurs :

```
hexagone::hexagone()
{   h[0]=point( 1,0,"A"); h[1]=point( 1,1,"B"); h[2]=point(0, 2,"C");
    h[3]=point(-1,1,"D"); h[4]=point(-1,0,"E"); h[5]=point(0,-1,"F");
}

hexagone::hexagone(point A, point B, point C, point D, point E, point F)
{   // ATTENTION : On suppose que tous les points sont différents !!
    h[0] = A; h[1] = B; h[2] = C; h[3] = D; h[4] = E; h[5] = F;
    strcpy(NH, A.nom()); strcat(NH, B.nom()); strcat(NH, C.nom());
    strcat(NH, D.nom()); strcat(NH, E.nom()); strcat(NH, F.nom());
}
```

2. On dit que deux hexagones (A, B, C, D, E, F) et 'A', B', C', D', E', F') sont équivalents s'il existe une permutation circulaire σ telle que : soit A' = σ(A), …, F' = σ(F), soit A' = σ(F), …, F' = σ(A). Par exemples, les haxagones (A, B, C, D, E, F) et (B, C, D, E, F, A) sont équivalents, de même que (A, B, C, D, E, F) et (F, E, D, C, B, A). En revanche, les hexagones (A, B, C, D, E, F) et (B, A, C, D, E, F) ne le sont pas. Précisez ce que recouvre géométriquement cette notion d'équivalence. Surchargez l'opérateur == pour en faire une fonction membre de la classe hexagone testant si deux hexagones sont équivalents. Nous dirons par la suite que deux hexagones sont égaux pour dire qu'ils sont équivalents, et différents pour dire qu'ils ne sont pas équivalents.

De manière géométrique, cette notion correspond à la conservation de l'hexagone. Quelque soit l'hexagone de départ, l'hexagone d'arrivée est exactement le même, géométriquement, à la différence près que les points ont un autre nom.

```
bool hexagone::operator==(hexagone H2)
{   int i,j,prems=-1; bool V1=true, V2=true;
    for (i=0; i<6; i++)                          // on recherche un point identique
        if (h[0] == H2[i])       prems = i;      // aux deux hexagones
    if (prems<0)  return false;   // s'il n'y a pas de points communs, ils ne peuvent
                                  // être égaux...
    for (i=0; i<6; i++)
    {   j = (prems+i > 5 ? (prems+i)-6 : prems+i);
        if (!(h[i] == H2[j]))    V1 = false;
    }
    for (i=0; i<6; i++)
    {   j = (prems-i < 0 ? (prems-i)+6 : prems-i);
        if (!(h[i] == H2[j]))    V2 = false;
    }
    return (V1 || V2);
}
```

3. Dans l'hexagone (A, B, C, D, E, F), on dit que les côtés (A, B) et (D, E) sont opposés : dans un hexagone, on a ainsi 3 paires de côtés opposés, lesquels ? On dit qu'un hexagone est un hexamy si et seulement si les côtés opposés sont deux à deux concourants et si les trois points de concours sont alignés (voir un exemple ci-dessous). Ecrivez une fonction C++ testant si un hexagone est un hexamy.

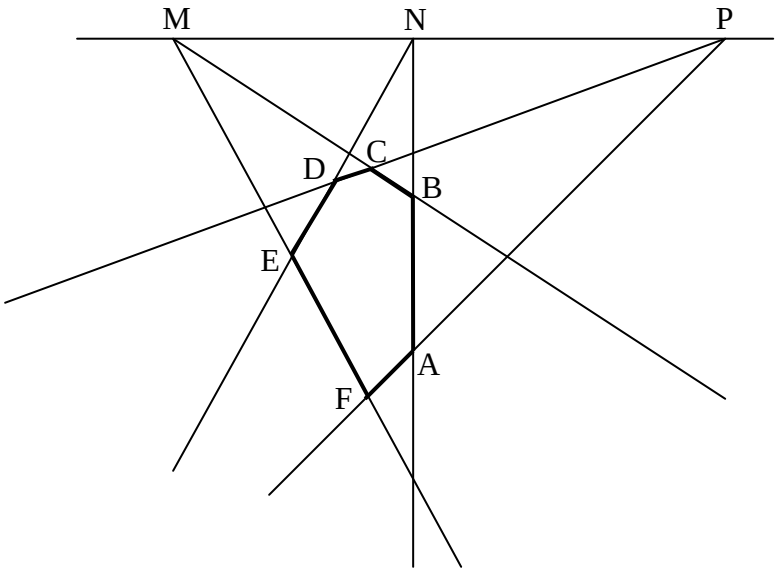


Figure 1 – Exemple d'hexamy

Dans un `hexagone` H, les trois côtés opposés sont alors : (`H.h[1]`, `H.h[2]`) & (`H.h[4]`, `H.h[5]`) ; (`H.h[2]`, `H.h[3]`) & (`H.h[5]`, `H.h[6]`) et enfin (`H.h[3]`, `H.h[4]`) & (`H.h[6]`, `H.h[1]`).

La fonction C++ qui testera si un `hexagone` est un hexamy sera une fonction membre de notre classe `hexagone` :

```
bool hexagone::est_un_hexamy()
{   point A,B,C;
    A = point(droite(h[0],h[1]), droite(h[3],h[4]));
    B = point(droite(h[1],h[2]), droite(h[4],h[5]));
    C = point(droite(h[2],h[3]), droite(h[5],h[0]));
    return alignes(A,B,C);
}
```

4. On admet le théorème suivant : **Si (A, B, C, D, E, F) est un hexamy, alors pour toute permutation σ, (σ(A), σ(B), σ(C), σ(D), σ(E), σ(F)) est aussi un hexamy.**

a) Montrez qu'à partir d'un hexamy, disons (A, B, C, D, E, F), on peut produire, par permutation des sommets, 60 hexamys *différents* (au sens des hexagones) et qu'on peut les choisir de sorte qu'ils commencent tous par le sommet A.

Il y a 6! permutations possibles des sommets. La relation d'équivalence == introduite plus haut (il est facile de vérifier qu'on a bien une relation d'équivalence) groupe tous les hexagones produits par paquets de 12. On a donc 6!/12 = 60 classes d'équivalence suivant ==, c'est-à-dire 60 hexamys différents. En utilisant une permutation circulaire, on peut prendre comme représentant des classes l'un des deux hexagones commençant par le point A.

b) En utilisant des *listes* de 6 points, donner un algorithme en pseudo-code indiquant comment obtenir toutes les permutations possibles de ces 6 points. En déduire une fonction permettant d'afficher toutes les permutations laissant le premier point à la première place. Est-ce que toutes les permutations produites sont différentes ? Comment y remédier ? (Une idée, mais ce n'est pas forcément la meilleure, est d'énumérer les permutations en disant qu'un point qui a été en première position le plus possible ne peut plus apparaître en dernière position, pourquoi ?)

On suppose programmée (c'est évident à faire) une fonction de prototype `listpoint hexagone::hexa2list()` produisant « la » liste des 6 points d'un hexagone. On a alors le pseudo-code récursif suivant:

```
void affiche_perm(listpoint LP)
{   listpoint RES;
    affiche_perm_bis(L, RES);
}

void affiche_perm_bis(listpoint LP, listpoint RES)
{   si LP est vide, alors afficher RES et passer à la ligne
    sinon
    pour i allant de 1 à (longueur de LP)
        e = LP[i];
        LP.r_ieme(i);
        RES.a_ieme(e,1);
        afficher_perm_bis(LP, res);
        L.a_ieme(e,i);
        RES.r_ieme(1);
    fait
}
```

Ce pseudo-code se traduit immédiatement en C++. Remarquons que cet algorithme affiche toutes les permutations d'une liste quelque soit sa longueur. Une version plus efficace (et plus simple à comprendre) avec 6 boucles `for` imbriquées s'en déduit directement :

```
void permut(listpoint LP)   // LP a exactement 6 points
{   point A, B, C, D, E, F;
    listpoint Lcopy=LP;
    for(int i=1; i<=6; i++)
    {   A=LP[i]; LP.r_ieme(i);
        for (int j=1; j<=5; j++)
        {   B=LP[j]; LP.r_ieme(j);
            for (int k=1; k<=4; k++)
            {   C=LP[k]; LP.r_ieme(k);
                for (int m=1; m<=3; m++)
                {   D=LP[m]; LP.r_ieme(m);
                    for (int n=1; n<=2; n++)
                    {   E=LP[n]; LP.r_ieme(n); F=LP[1];
                        cout <<A<<","<<B<<","<<C<<","<<D<<","<<E<<","<<F<<endl;
                        LP.a_ieme(E, n);
                    }
                    LP.a_ieme(D, m);
                }
                LP.a_ieme(C, k);
            }
            LP.a_ieme(B, j);
        }
        LP.a_ieme(A, i);
    }
}
```

Pour appliquer cet algorithme à notre problème, il suffit :

- *(i)* soit pour les boucles imbriquées : d'enlever la première boucle (`A=L[1]`) et de faire démarrer les autres à 2 ;
- *(ii)* soit pour la version récursive : de laisser le premier élément en place et de modifier légèrement le code de `affiche_perm` et `affiche_perm_bis` en :

```
void affiche_perm_s1(listpoint LP)
{   listpoint RES;
    point e=LP[i];
    LP.r_ieme(1);
    affiche_perm_bis_s1(e, L, RES);
}

void affiche_perm_bis_s1(point f, listpoint LP, listpoint RES)
{   si LP est vide, alors afficher e puis RES et passer à la ligne
```

```
    sinon
    pour i de 1 à longueur de L
        e=L[i]
        L.r_ieme(i)
        RES.a_ieme(e,1)
        afficher_perm_bis_s1(f,LP,RES)
        L.a_ieme(e,i)
        RES.r_ieme(1)
    fait
}
```

Mais alors on affichera un même hexagone sous deux formes équivalentes : l'une en comptant les points dans un sens et l'autre en comptant dans l'autre sens. Une idée pour éviter ces redondances consiste à dire que si on produit toutes les permutations sur 5 points commençant par B (il y en a 4! = 24) alors il ne faut pas produire celles se terminant par B car elles sont équivalentes aux premières. On produira ensuite toutes les permutations commençant par C et ne se terminant pas par B : il y a en 4! – 3! = 18 ou 3 × 3!, puis celles commençant par D et ne se terminant ni par B, ni par C, il y en a 2 × 3! = 12 et finalement celles commençant par E et ne se terminant ni par B, ni par C, ni par D : elles se terminent donc obligatoirement par F et il y en a 3! : on retrouve bien les 60 permutations. On peut attaquer directement ce problème en considérant le cas des listes à 5 points comme indiqué : je vais faire ici une méthode générale qui fonctionne avec un nombre quelconque de points. Décomposons l'algorithme qui en résulte en plusieurs fonctions. Observons tout d'abord que la petite variante du pseudo-code précédent (avec réutilisation de `afficher_perm_bis()`)

```
    void affiche_perm_n(listpoint LP, int n)
    {   listpoint RES;
        pour i de n à longueur de LP
            e=LP[i]
            LP.r_ieme(i)
            RES.a_ieme(e,1) // mettre en dernière position un point hors des n premiers
            afficher_perm_bis(LP, RES)
            LP.a_ieme(e,i)
            RES.i_eme(1)
        fait
    }
```

produit toutes les permutations de L qui n'ont aucun des *n* premiers points en dernière position. En utilisant cette idée, on a :

```
void affiche_perm_equiv(listpoint LP)   // LP est une liste à 5 éléments
{   affiche_perm_s1(LP)
    pour i de 2 à LP.longueur()-1
        e=LP[i]
        LP.r_ieme(i)
        affiche_perm1(e, LP, i)
        LP.a_ieme(e, i)
    fait
}
```

avec :

```
void affiche_perm1(point f, listpoint LP, int n)
{   listpoint RES;
    pour i de n à longueur de LP
        e = LP[i]
        LP.r_ieme(i)
        RES.a_ieme(e,1)
        afficher_perm_ter(f, LP, RES)
        LP.a_ieme(e, i)
        RES.r_ieme(i)
    fait
```

```
}

void affiche_perm_ter(point f, listpoint LP, listpoint RES)
{   si LP est vide, alors afficher f puis RES et passer à la ligne
    sinon
    pour i de 1 à longueur de L
        e=L[i]
        L.r_ieme(i)
        RES.a_ieme(e,1)
        afficher_perm_ter(f, LP, RES)
        LP.a_ieme(e, i)
        RES.r_ieme(1)
    fait
}
```

c) Ecrivez une fonction permettant d'afficher tous les hexamys non équivalents possibles à partir d'un hexamy donné. Ecrivez une fonction C++ testant sur un exemple, le théorème énoncé ci-dessus.

Voici le code C++ faisant la synthèse de tout ce qui a été raconté plus haut (encore une fois, il y a une solution plus simple, mais moins générale, utilisant des boucles) :

```
void affiche_hexa(hexagone H)
{   listpoint LP = H.hexa2list();
    point A = LP[1];
    LP.r_ieme(1);
    affiche_perm_equiv(A, LP);
}

void affiche_perm_equiv(point A, listpoint LP)
{   point e = LP[1];
    listpoint RES;
    LP.r_ieme(1);
    affiche_perm_ter(A, e, LP, RES);
    LP.a_ieme(e, 1) ;
    for (int i=2; i<LP.longueur(); i++)
    {   e = LP[i];
        LP.r_ieme(i);
        affiche_perm1(A, e, LP, i);
        LP.a_ieme(e, 1);
    }
}

void affiche_perm1(point A, point f, listpoint LP, int n)
{   listpoint RES;
    for (int i=n; i<=LP.longueur(); i++)
    {   e = LP[i];
        LP.r_ieme(i);
        RES.a_ieme(e, 1);
        afficher_perm_ter(A, f, LP, RES);
        LP.a_ieme(e, i);
        RES.r_ieme(1);
    }
}

void affiche_perm_ter(point A, point f, listpoint LP, listpoint RES)
{   if (L.vide())  cout <<A<<","<<f<<","<<res<< endl;  // (*) Affichage, à modifier
    else                                               // pour le test hexamy
    for (int i=1; i<=LP.longueur(); i++)
    {   e = LP[i];
        LP.r_ieme(i);
        RES.a_ieme(e, 1);
        afficher_perm_ter(f, LP, RES);
        LP.a_ieme(e, i);
        RES.r_ieme(i);
    }
```

```
}
```

Pour écrire une fonction C++ testant sur un exemple le théorème énoncé ci-dessus, c'est très simple : il suffit au moment de l'affichage (voir `(*)` dans le code plus haut) de reconstruire un `hexagone` à partir des points et de faire le test.

5. On veut généraliser les idées des questions 1 et 2 aux polygones (ayant leurs sommets distincts). Proposez une implantation de la classe `polygone`. Précisez, en particulier, l'implantation de l'opérateur `==` testant l'équivalence de deux polygones.

```
// fichier hexamy.h
class polygone
{    point p[100];
     int   NDP;      // correspond au nombre de points du polygone
     char  NP[32];
    public:
           polygone();
           polygone(point centre, int ndp); // polygone régulier centré (exercice)
           polygone(listpoint E);
     char* nom();                          // retourne le nom du polygone
     int   NbreCotes();                    // retourne le nombre de côtés du polygone
     point operator[](int i);              // accès au (i-1)-ème point de l'haxagone
     void  modification(point P, int pos); // modification (avec vérification)
     void  a_ieme(point P, int pos);       // ajoute P à la (pos-1)-ème position
     void  r_ieme(int pos);                // retire le (pos-1)-ème point du polygone
     void  get();                          // saisie d'un polygone
     bool  est_un_hexamy();                // teste si un polygone est un hexamy
     bool  operator==(polygone P2);        // test d'équivalence
    friend ostream& operator<<(ostream& o, polygone H);
}; // fin de la classe polygone
```

On remarquera que les hexagones font parties des polygones, d'où la présence de la fonction **`bool`** `est_un_hexamy()`, qui teste si un polygone (à 6 côtés $\Rightarrow$ `NDP == 6`) est un `hexamy`. On a aussi rajouté les deux fonctions d'ajout d'un élément et de retrait, car un polygone n'est pas limité en nombre de côtés.

Implémentation de la fonction **`operator`**`==` :

```
bool polygone::operator==(polygone P2)
{   int i,j,prems=-1; bool V1=true, V2=true;
    if (NDP != P2.NDP)    return false;

    for (i=0; i<NDP; i++)                        // on recherche un point identique
        if (p[0] == P2[i])       prems = i;      // aux deux polygones
    if (prems<0)  return false;   // s'il n'y a pas de points communs, ils ne peuvent
                                  // être égaux...
    for (i=0; i<NDP; i++)
    {   j = (prems+i >= NDP ? (prems+i)-NDP : prems+i);
        if (!(p[i] == P2[j]))    V1 = false;
    }
    for (i=0; i<NDP; i++)
    {   j = (prems-i < 0 ? (prems-i)+NDP : prems-i);
        if (!(p[i] == P2[j]))    V2 = false;
    }
    return (V1 || V2);
}
```

On remarquera que la classe `polygone` est la même que la classe `hexagone`, à part les noms qui ont changé, et modulo quelques fonctions qui ont été rajoutées, fonctions jugées utiles à la classe `polygone`. La fonction **`operator`**`==` est également la même, mais on a dû rajouter un test sur le nombre de points.