Des indications concernant quelques fonctions utiles de <math.h> sont données en annexe à la fin du document.

I) Classe BasicAngle

La classe BasicAngle est une classe de base pour représenter des angles.

```
class BasicAngle
{
…
private :
    double val;          valeur de l'angle en radians entre –π et +π
};
```

1) Ecrire un constructeur sans paramètre (angle nul) et un constructeur recevant la valeur de l'angle. Cette valeur peut être quelconque, le constructeur doit la ramener à son équivalent dans la plage [-π, +π] en appliquant l'algorithme : faire le modulo de la valeur par 2π (fonction fmod en annexe), puis retirer 2π si le résultat est > π ou ajouter 2π s'il est < -π.

2) Ecrire un opérateur << d'affichage.

3) Ajouter une fonction *double getRadian() const* et *void setRadian(double v)* pour récupérer ou modifier la valeur de l'angle (*v* quelconque, à convertir en son équivalent dans [-π, +π]).

II) Classe Angle

La classe Angle hérite de BasicAngle, elle ajoute des fonctions permettant d'exprimer des angles en degrés, des opérateurs et des fonctions trigonométriques.
Elle ne stocke pas de donnée supplémentaire par rapport à BasicAngle.

Définir :
```
enum Unite {RAD, DEG};          // radians, degrés
```

1) Expression des angles en radians ou degrés :
   a) Ecrire un constructeur sans paramètre (angle nul) et un constructeur permettant de créer un angle à partir d'une valeur en radians ou en degrés, conformément aux exemples suivants :
   Angle a1(4.18, RAD); // 4.18 radians
   Angle a2(60.5, DEG); // 60.5 degrés
   Angle a3(0.94); // par défaut valeur en radians
   b) Ajouter les fonctions *double getDegre() const* et *void setDegre(double v)* pour récupérer ou modifier la valeur de l'angle, exprimée en degrés.
   c) Ecrire une fonction *void afficher(Unite u) const* affichant la valeur de l'angle dans l'unité *u*. Prévoir RAD comme valeur par défaut pour l'unité.

2) Opérateurs :
   a) Créer l'opérateur + d'addition de deux Angle.
   b) Ajouter un opérateur * de multiplication d'un Angle par un réel.

3) Fonctions trigonométriques :
   a) Ecrire des fonctions *double sinus() const*, *double cosinus() const* et *double tangente() const* retournant le sinus, le cosinus et la tangente de l'angle.
   b) Ecrire des fonctions statiques *Angle arcsin(double x)*, *Angle arccos(double x)* et *Angle arctan(double x)* retournant un objet Angle représentant l'arc sinus, l'arc cosinus et l'arc tangente de x, et *Angle arctan2(double y, double x)* équivalente à la fonction atan2 de <math.h>.

III) Classe Point

Le but est de créer la classe Point suivante qui stocke les coordonnées cartésiennes mais qui permet de manipuler un point par ses coordonnées cartésiennes ou polaires.

```
class Point
{
…
private :
   double x, y;
};
```

1) Constructeurs :
   Ecrire un constructeur par défaut (point 0, 0), un constructeur recevant les coordonnées cartésiennes (deux double) et un constructeur recevant les coordonnées polaires (un double pour le rayon – que l'on supposera positif, le cas d'un rayon négatif n'est pas à traiter - et un objet Angle pour l'angle).

2) Fonctions "get" et "set" :
   a) Ecrire les fonctions *double getX() const* et *double getY() const* retournant les coordonnées cartésiennes, et *double getR() const* et *Angle getT() const* retournant les coordonnées polaires.

b) Ajouter les fonctions suivantes permettant de modifier le point en agissant sur l'une de ses coordonnées (en conservant l'autre coordonnée constante) : *void setX(double newX)* et *void setY(double newY)* pour les coordonnées cartésiennes, et *void setRl(double newR)* et *void setT(const Angle& newT)* pour les coordonnées polaires.

3) Affichage :
Ecrire la fonction *void afficher(TypeCoord type_coord) const* qui affiche les coordonnées du points, soit les cartésiennes, soit les polaires en fonctions du paramètre type_coord.
Pour *TypeCoord* définir un enum :
enum TypeCoord {CART, POL}; // cartésiennes, polaires
Pour le paramètre type_coord prévoir CART comme une valeur par défaut.

4) Déplacement et distance :
a) Ecrire l'opérateur + ajoutant un déplacement à un point, le résultat étant un point.
Pour exprimer un déplacement, définir (dans Point.h) la structure :

```
struct Depl {
   double dx, dy;          // valeurs à ajouter à x et à y
};
```

Ecrire aussi l'opérateur += associé.
Exemple :

```
Point p1(3, 5);
Depl d;
d.dx = 8;
d.dy = 9;
Point p2 = p1 + d;      // p2 est le point (11, 14)
p2 += d;                // p2 est le point (19, 23)
```

b) Ecrire l'opérateur – qui fait la différence entre deux points, le résultat étant un déplacement.
Exemple :

```
Point p1(7, 2), p2(4, 3);
Depl d = p1 – p2;       // d vaut {3, -1}
```

c) Ajouter une fonction calculant la distance entre deux points.

IV) Classe Chemin

Cette classe modélise un chemin, représenté par une suite de points stocké dans un vecteur. Pour rappel, il n'est pas nécessaire d'utiliser un itérateur pour parcourir un vecteur[1].

[1] Si vous souhaitez malgré tout le faire dans une fonction membre qui n'est pas autorisée à modifier l'objet courant (const), vous devez utiliser un itérateur de format const_iterator.

```
class Chemin
{
…
private :
    vector<Point> points;
};
```

1) Ecrire un constructeur *Chemin(const char* nomFic, TypeCoord type_coord)* qui crée un chemin à partir d'un fichier texte qui contient la suite des coordonnées des points.
   Le paramètre type_coord indique si les coordonnées du fichier sont des coordonnées cartésiennes ou polaires (avec l'angle en radian).

2) Ecrire une fonction *void afficher(TypeCoord type_coord) const* qui affiche le chemin, c'est-à-dire la suite des points, soit les coordonnées cartésiennes, soit les coordonnées polaires.

3) Ajouter une fonction qui retourne la longueur du chemin, c'est dire la somme des longueurs des segments constitués par la suite des points.

4) Ajouter une fonction *void translation(const Point& nouvDepart)* qui effectue une translation du chemin. Le paramètre nouvDepart indique le nouveau premier point. Les autres points doivent subir la même translation.

5) Ecrire une fonction qui retourne le rectangle délimitant au plus près le chemin.
   Le rectangle est retourné sous forme d'une structure Rect (à définir dans Chemin.h) représentant un rectangle de côté parallèles aux axes de coordonnées.

```
struct Rect {
    double gauche, droite, bas, haut;        // coordonnées des côtés gauche, droit, bas et haut
};
```

Annexe :

Inclure <math.h>.

- M_PI : constante pour π (certains compilateurs ne reconnaissent pas cette constante, dans ce cas introduire une constante symbolique).
- INFINITY : constante représentant l'infini pour les réels (> tout nombre de type *float* ou *double*).
- double fmod (double x, double y) : modulo du nombre réel x par le nombre réel y.
- double sin(double a), double cos(double a) et double tan(double a) : sinus, cosinus et tangente d'un angle a exprimé en radians.
- double asin(double x), double acos(double x) et double atan(double x) : arc sinus, arc cosinus et arc tangente de x exprimés en radians.
- double atan2(double y, double x) : arc tangente de y/x, c'est-à-dire l'angle des coordonnées polaires de (x, y), exprimé en radians.
- double sqrt(double x) : racine carrée de x.