

Enregistrements (Corrigé TD1)

1. Points

- Définir sous forme d'enregistrement le type **point** composé de deux réels **x** et **y** ;
- Ecrire la fonction **saisirPoint** qui demande à l'utilisateur de saisir au clavier les coordonnées d'un point, puis qui retourne ce point ;
- Ecrire la fonction **distancePoints** qui prend en entrée deux points et retourne la distance entre ces points (faire appel au sous-programme **racine_carree** qui retourne la racine carrée d'un nombre). Rappel: la distance entre deux points $(x1,y1)$ et $(x2,y2)$ est la racine carrée de $((x1-x2)^2 + (y1-y2)^2)$.
- Ecrire le programme principal qui, en faisant appel aux fonctions précédentes, fait saisir deux points et affiche la distance qui les sépare.

structure point

```
{  
    x,y: réel  
}
```

fonction saisirPoint(): point

```
{  
    p: point  
  
    afficher("Entrez les coordonnées")  
    saisir(p.x, p.y)  
    retourner p  
}
```

fonction distancePoints(E p1, p2: point): réel

```
{  
  
    retourner racine_carree((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y))  
}
```

Distance entre deux points //Programme principal

```
{  
    A, B: point  
    D: réel  
  
    // Saisie des points  
    A ★ saisirPoint()  
    B ★ saisirPoint()  
  
    D ★ distancePoints(A, B)  
    afficher("Distance : ", D) //ou afficher("Distance : ", distancePoints(A, B))  
}
```

2. Disques

- Définir sous forme d'enregistrement le type **disque** composé de deux champs **rayon** (type réel) et **centre** (type point);
- Ecrire la fonction **saisirDisque** qui demande à l'utilisateur de saisir au clavier les coordonnées du centre d'un disque puis son rayon, et qui retourne ce disque;
- Ecrire la fonction **testInclusion** qui prend en entrée un point et un disque et qui détermine si le point est inclus dans le disque ;
- Ecrire le programme principal qui, en faisant appel aux fonctions précédentes, fait saisir un point et un disque et affiche un message indiquant si le premier est inclus dans le second.

structure disque

```
{  
    rayon: réel  
    centre: point  
}
```

fonction saisirDisque(): disque

```
{  
    d: disque  
  
    afficher("Entrez les coordonnées de centre : ")  
    d.centre ★ saisirPoint()    // saisir(d.centre.x,d.centre.y)  
    afficher("Entrez le rayon : ")  
    saisir(d.rayon)  
    retourner d  
}
```

fonction testInclusion(E p: point, E d: disque): booléen

```
{  
    dist: réel  
  
    dist ★ distancePoints(p, d.centre)  
    si (dist < d.rayon) alors retourner vrai  
    sinon retourner faux  
}
```

Inclusion point-cercle //Programme principal

```
{  
    p: point  
    d: disque  
  
    p ★ saisirPoint()  
    d ★ saisirDisque()  
    si (testInclusion(p, d)) alors afficher("Le point est inclus dans le disque")  
    sinon afficher("Le point n'est pas inclus dans le disque")  
}
```

3. Polygones

- Définir sous forme d'enregistrement le type **polygone** composé de deux champs **nombre_sommets** (type entier) et **sommets** (un tableau de points). Remarques :
 - Par défaut on considère des polygones *fermés* – le dernier sommet est relié au premier ;
 - On suppose qu'un polygone ne peut avoir plus de 100 sommets.
- Ecrire la fonction **saisirPolygone** qui demande à l'utilisateur de saisir au clavier un nombre de sommets, puis les coordonnées des sommets d'un polygone retourné par la suite.
- Ecrire la fonction **perimetre** qui prend en entrée un polygone et qui retourne son périmètre, c-à-d la somme des distances entre le premier sommet et le deuxième, le deuxième et le troisième, ... , le dernier et le premier.
- Ecrire le programme principal qui, en faisant appel aux fonctions précédentes, fait saisir les coordonnées d'un polygone et affiche son périmètre.

structure polygone

```
{  
    nombre_sommets : entier  
    sommets: vecteur de 100 point  
}
```

fonction saisirPolygone(): polygone

```
{  
  
    p: polygone  
    i: entier  
  
    afficher("Entrez le nb de sommets :")  
    saisir(p.nombre_sommets)  
    pour i de 0 à p.nombre_sommets - 1 faire  
    {  
        afficher("Sommet N° ",i," : ")  
        p.sommets[i] ★ saisirPoint()  
    }  
    retourner p  
}
```

fonction périmètre(E p: polygone): réel

```
{  
    i: entier  
    périm: réel  
  
    périm★ distance(p.sommets[0], p.sommets[p.nombre_sommets - 1]) // Initialisation  
    pour i de 0 à p.nombre_sommets - 2 faire  
        périm★ périm+ distance(p.sommets[i], p.sommets[i+1])  
    retourner périm  
}
```

```

Périmètre de polygone      //Programme principal
{
    poly: polygone
    poly ★ saisirPolygone()
    afficher("Périmètre: ", périmètre(poly))
}

```

4. Dates

- Définir sous forme d'enregistrement le type **date** composé de trois entiers **jour**, **mois** et **annee**.
- Ecrire la fonction **testDates** qui prend en entrée deux dates **d1** et **d2** et qui retourne:
 - **1** si **d1** est antérieure à **d2**
 - **0** si **d1** est égale à **d2**
 - **-1** si **d1** est postérieure à **d2**
- Ecrire la procédure **afficheDatesOrdre** qui prend en entrée un tableau de dates et sa taille, et qui, en faisant appel à la fonction précédente, doit afficher le contenu du tableau dans l'ordre chronologique.
- Ecrire le programme principal qui génère aléatoirement un tableau de 10 dates (utiliser la fonction **alea**) puis fait appel à la fonction précédente pour afficher ces dates dans l'ordre.

structure date

```

{
    jour, mois, annee: entier
}

```

fonction testDates(E d1,d2: date): entier

```

{
    si (d1. annee < d2. annee) alors retourner 1
    sinon si (d1. annee > d2. annee) alors retourner -1
        sinon si (d1. mois < d2. mois) alors retourner 1 // sinon : égalité des années
            sinon si (d1. mois > d2. mois) alors retourner -1
                sinon si (d1. jour < d2. jour) alors retourner 1 // sinon : égalité des mois
                    sinon si (d1. jour > d2. jour) alors retourner -1
                        sinon retourner 0 // sinon : égalité des jours
}

```

procédure afficheDatesOrdre(E vd: vecteur de date; E taille: entier)

```

{
    min,tmp : date
    i,j, ind_min : entier

    // Tri des dates dans l'ordre chronologique
    pour j de 0 à taille-2 faire
        {

```

```

        min ★ vd[j]    // min = la 1ère date
        ind_min ★ j
        pour i de j+1 à taille-1 faire
            si (testDates(vd[i],min) = 1) alors
                {
                    min ★ vd[i]
                    ind_min ★ i
                }
            si (ind_min ⊕ j) alors // permutation
                {
                    tmp ★ vd[j]
                    vd[j] ★ vd[ind_min]
                    vd[ind_min] ★ tmp;
                }
        }

// Affichage dans l'ordre
pour i de 0 à taille-1 faire afficher (vd[i].jour, "/", vd[i].mois, "/", vd[i].annee)
}

```

```

Trier dates    //Programme principal
{
    vd: vecteur de 10 dates
    i,nbj: entier

    pour i de 0 à 9 faire    // Initialisation du tableau
    {
        vd[i].annee ★ alea()*2010
        répéter vd[i].mois ★ alea()*12 jusqu'à (vd[i].mois ≠ 0)
        Selon (vd[i].mois)
        {
            1, 3, 5, 7, 8, 10, 12: nbj ★ 31
            4, 6, 9, 11: nbj ★ 30
            2: nbj ★ 28
        }
        répéter vd[i].jour ★ alea()*nbj jusqu'à (vd[i].jour ≠ 0)
    }
    afficheDatesOrdre(vd,10)
}

```