# Algorithmique
# et
# Bases de la programmation

Durée : *1h55*
Documents autorisés : **AUCUN**

L'objectif de cet examen pratique est de développer une librairie C proposant le type `Polyligne`.

## Informations

### Avant de commencer l'examen

Connectez vous à la machine et suivez les instructions suivantes :
- copiez l'archive `C.tgz` qui se trouve dans `/opt/files` dans votre répertoire (commande `cp`) ;
- après l'avoir décompressée (commande `tar` avec les options `zxvf`), renommez le répertoire en lui donnant comme nom : 'EXAM-ALGO-' suivi de votre nom et prenom, sans espace et en minuscule sauf pour les premières lettres du prénom et du nom. Si je devais passer l'examen, je nommerai le répertoire EXAM-ALGO-DelestreNicolas (je taperai donc dans le terminal `mv C EXAM-ALGO-DelestreNicolas`)

### À la fin de l'examen

Cinq minutes avant la fin de l'examen :
- assurez vous que votre projet compile ;
- créez une archive de votre répertoire (commande `tar` avec les options `zcvf`), que vous nommerez comme le répertoire (avec le préfixe `.tgz`) ;
- déposez sur moodle votre archive (cours ≪ base de la programmation et algorithmique ≫, section ≪ examens pratiques ≫).

**Attention :**
- le lien de dépôt est actif uniquement 10 minutes, si vous dépassez le délai, votre note sera 0 ;
- on ne peut déposer qu'une seule fois le projet sur moodle.

### Quelques conseils

Pour réussir votre examen suivez les conseils suivants :
- respectez bien les consignes données ci dessus. **Le fait de ne pas les respecter vous fera perdre 2 points sur la note finale** ;
- par défaut le projet compile et s'exécute. Après le développement de chaque fonction C, compilez votre projet (`make`) et ne passez à la fonction suivante que lorsque votre projet compile : **le fait de rendre un projet qui ne compile pas ou ne s'exécute pas (par exemple à cause d'un *segmentation fault*), divisera par deux votre note finale** ;
- la note finale sera fonction :
  - du nombre de tests unitaires qui seront valides (exécution du programme `bin/testPolyligne_CUnit`) ;
  - de la quantité, de la qualité et de la lisibilité du code (indentation et nom des identifiants).

# 1 Contexte : Le type `Point2D`

## 1.1 Analyse

Soit le TAD `Point2D` suivant :

| | | |
|---|---|---|
| **Nom**: | Point2D | |
| **Utilise**: | **Reel** | |
| **Opérations**: | point2D: | **Reel** × **Reel** → Point2D |
| | obtenirX: | Point2D → **Reel** |
| | obtenirY: | Point2D → **Reel** |
| | distanceEuclidienne: | Point2D × Point2D → **Reel** |
| | translater: | Point2D × Point2D → Point2D |
| | faireRotation: | Point2D × Point2D × **Reel** → Point2D |

## 1.2 Conception préliminaire

Dans le paradigme de la programmation structurée, le TAD `Point2D` peut être utilisé à l'aide des fonctions et procédures suivantes : :

- **fonction** point2D **(**x,y : **Reel) :** Point2D
- **fonction** obtenirX **(**p : Point2D**) : Reel**
- **fonction** obtenirY **(**p : Point2D**) : Reel**
- **fonction** distanceEuclidienne **(**p1,p2 : Point2D**) : Reel**
- **procédure** translater **(E/S** p : Point2D,**E** vecteur : Point2D**)**
- **procédure** faireRotation **(E/S** p : Point2D,**E** centre : Point2D, angle : **Reel)**

## 1.3 En C

En C, nous pouvons manipuler des variables de type `PT_Point2D` en utilisant la librairie *lib/libPoint2D.a* (construite à partir des fichiers `Point2D.h` et `Point2D.c`), qui propose entre autres les fonctions C suivantes (extrait du fichier `Point2D.h`)[1] :

```
1 /* Partie publique */
2
3 /* Partie métier */
4 PT_Point2D PT_point2D(float x,float y);
5 float PT_obtenirX(PT_Point2D pt);
6 float PT_obtenirY(PT_Point2D pt);
7 float PT_distanceEuclidienne(PT_Point2D pt1, PT_Point2D pt2);
8 void PT_translater(PT_Point2D *ppt, PT_Point2D vecteur);
9 void PT_faireRotation(PT_Point2D *ppt, PT_Point2D centre, float angle);
10 int PT_egaux(PT_Point2D pt1,PT_Point2D pt2);
11
12 /* Partie pour pouvoir manipuler des collections de Point2D */
13 /* Fonctions compatibles avec celles définies dans Collection.h */
14 void* PT_dupliquer(void*);
15 void PT_liberer(void*);
16 int PT_comparer(void*,void*);
```

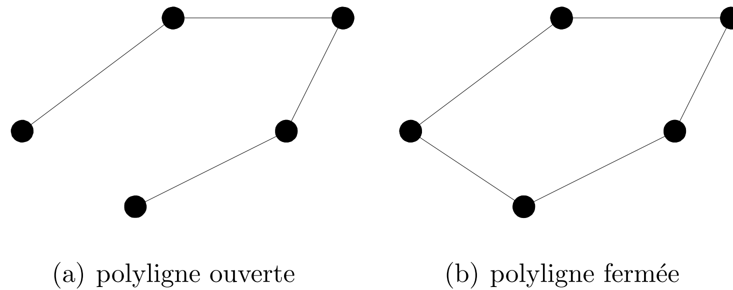
1. Les dernières fonctions sont utilisées dans l'utilisation d'un point 2D dans une collection générique, comme nous l'avons vu en cours

# 2 Le TAD `Polyligne`

## 2.1 Définition

« Une ligne polygonale, ou ligne brisée (on utilise aussi parfois polyligne par traduction de l'anglais *polyline*) est une figure géométrique formée d'une suite de segments, la seconde extrémité de chacun d'entre eux étant la première du suivant.[...] Un polygone est une ligne polygonale fermée. » (Wikipédia)

La figure 1 présente deux polylignes composées de 5 points.



(a) polyligne ouverte (b) polyligne fermée

FIGURE 1 – Deux polylignes

De cette définition nous pouvons faire les constats suivants :

- Une polyligne est constituée d'au moins deux points ;
- On peut obtenir le nombre de points d'un polyligne ;
- Une polyligne est ouverte ou fermée (qu'elle soit ouverte ou fermée ne change pas le nombre de points) ;
- On peut ajouter, supprimer des points à une polyligne (par exemple la figure 2 présente la supression du troisième point de la polyligne ouverte de la figure 1) ;
- On peut parcourir les points d'une polyligne ;
- On peut effectuer des transformations géométriques (translation, rotation, etc.) ;
- On peut calculer des propriétés d'une polyligne (par exemple sa longueur totale).

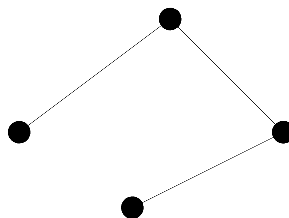


FIGURE 2 – Supression d'un point

## 2.2 Analyse

À partir de cette définition et des constats, nous pouvons définir le TAD suivant :

**Nom**: Polyligne
**Utilise**: Point2D,**Reel**,**Booleen**,**Naturel**
**Opérations**: polyligne: Point2D $\times$ Point2D $\times$ **Booleen** $\rightarrow$ Polyligne

| | | |
|---|---|---|
| | estFermee: | Polyligne $\rightarrow$ **Booleen** |
| | fermer: | Polyligne $\rightarrow$ Polyligne |
| | ouvrir: | Polyligne $\rightarrow$ Polyligne |
| | nbPoints: | Polyligne $\rightarrow$ **Naturel** |
| | iemePoint: | Polyligne $\times$ **Naturel** $\nrightarrow$ Point2D |
| | ajouterPoint: | Polyligne $\times$ Point2D $\times$ **Naturel** $\nrightarrow$ Point2D |
| | supprimerPoint: | Polyligne $\times$ **Naturel** $\nrightarrow$ Polyligne |
| | longueur: | Polyligne $\rightarrow$ **Reel** |
| | translater: | Polyligne $\times$ Point2D $\rightarrow$ Polyligne |
| | faireRotation: | Polyligne $\times$ Point2D $\times$ **Reel** $\rightarrow$ Polyligne |
| **Préconditions**: | iemePoint(pl,i): | $0 < i \leq nbPoints(pl)$ |
| | ajouterPoint(pl,pt,i): | $0 < i \leq nbPoints(pl) + 1$ |
| | supprimerPoint(pl,i): | $0 < i \leq nbPoints(pl)$ et $nbPoints(pl) \geq 3$ |

## 2.3 Conception préliminaire

Dans le paradigme de la programmation structurée, ces opérations se concrétisent par les fonctions et procédures suivantes :

- **fonction** polyligne **(**pt1,pt2 : Point2D, estFermee : **Booleen) :** Polyligne
- **fonction** estFermee **(**pl ; Polyligne**) : Booleen**
- **procédure** fermer **(E/S** pl : Polyligne**)**
- **procédure** ouvrir **(E/S** pl : Polyligne**)**
- **fonction** nbPoints **(**pl ; Polyligne**) : Naturel**
- **fonction** iemePoint **(**pl ; Polyligne, position : **Naturel) :** Point2D

  ⌊**précondition(s)** position>0 et position≤ nbPoints(pl)
- **procédure** ajouterPoint **(E/S** pl : Polyligne,**E** pt : Point2D, position : **Naturel)**

  ⌊**précondition(s)** position>0 et position≤ nbPoints(pl)+1
- **procédure** supprimer **(E/S** pl : Polyligne,**E** position : **Naturel)**

  ⌊**précondition(s)** nbPoints(pl)≥ 3 et position>0 et position≤ nbPoints(pl)
- **fonction** longueur **(**pl ; Polyligne**) : Reel**
- **procédure** translater **(E/S** pl : Polyligne,**E** vecteur : Point2D**)**
- **procédure** faireRotation **(E/S** pl : Polyligne,**E** centre : Point2D, angleEnRadian : **Reel)**

## 2.4 Conception détaillée

On propose de représenter le type Polyligne de la façon suivante :

**Type** Polyligne = **Structure**
lesPts : ListeChainee<Point2D>
nbPts : **Naturel**
estFermee : **Booleen**
**finstructure**

Les algorithmes de ce type sont simples et utilisent principalement des fonctions ou procédures de manipulation de liste chaînée, dont voici quelques algorithmes pour rappel :

1. Parcours itératif d'une liste chaînée (pour effectuer un certain traitement) :
   **procédure** parcourir **(E** l : ListeChainee**)**

   **Déclaration** temp : ListeChainee

**debut**
temp ← l
**tant que** non estVide(temp) **faire**
traiter(obtenirElement(temp))
temp ← obtenirListeSuivante(temp)
**fintantque**
**fin**

2. Parcours récursif d'une liste chaînée (pour effectuer un certain traitement) :
**procédure** parcourir **(E** l : ListeChainee**)**
**debut**
**si** non estVide(l) **alors**
traiter(obtenirElement(l))
parcourir(obtenirListeSuivante(l))
**finsi**
**fin**

3. Insérer un élément à la ième place d'une liste chaînée
**procédure** inserer **(E/S** l : ListeChainee, **E** place : **Naturel**, e : Element**)**
⌊**précondition(s)** place>0 et place≤ longueur(l)+1
**Déclaration** temp : ListeChainee
**debut**
**si** place=1 **alors**
ajouter(l,e)
**sinon**
temp ← obtenirListeSuivante(l)
inserer(temp,place-1,e)
fixerListeSuivante(l,temp)
**finsi**
**fin**

4. Supprimer un élément à la ième place d'une liste chaînée
**procédure** supprimer **(E/S** l : ListeChainee, **E** place : **Naturel)**
⌊**précondition(s)** place>0 et place≤ longueur(l)
**Déclaration** temp : ListeChainee
**debut**
**si** place=1 **alors**
l ← obtenirListeSuivante(l)
**sinon**
temp ← obtenirListeSuivante(l)
supprimer(temp,place-1)
fixerListeSuivante(l,temp)
**finsi**
**fin**

# 3 Développement

Nous décidons d'utiliser l'implantation générique en C vue en cours du TAD `ListeChainee` (utilisation du `void*` et des types de fonctions d'ajout, de suppression et de comparaison de deux éléments génériques).

Le projet C contient les fichiers suivants :

- `include/Point2D.h` : partie publique du type `PT_Point2D` ;
- `include/Polyligne.h` : partie publique du type `PL_Polyligne` ;
- `include/ListeChainee.h` : partie publique du type générique `LC_ListeChainee` ;
- `include/Collection.h` : définition des signatures des fonctions de copie, suppression et comparaison d'éléments d'une collection générique ; `LC_ListeChainee` ;
- `src/Point2D.c` : partie privée du type `PT_Point2D` ;
- `src/Polyligne.c` : partie privée du type `PL_Polyligne` ;
- `src/ListeChainee.c` : partie privée du type `LC_ListeChainee` ;
- `src/testPoint2D_CUnit.c` : code source d'un test unitaire du type `PT_Point2D`
- `src/testPolyligne_CUnit.c` : code source d'un test unitaire du type `PL_Polyligne`.

L'exécution du `make` produit :

- `lib/libPoint2D.a` : la librairie C proposant le type `PT_Point2D` ;
- `lib/libPolyligne.a` : la librairie C proposant le type `PL_Polyligne` ;
- `lib/libListeChainee.a` : la librairie C proposant le type `LC_ListeChainee` ;
- `bin/testPoint2D_CUnit` : test unitaire du type `PT_Point2D` ;
- `bin/testPolyligne_CUnit` : test unitaire de type `PL_Polyligne`.

**Travail à réaliser** Complétez le fichier `src/Polyligne.c`.