Examen final - LO01 P2012

2 H - Sans documents (sauf diagrammes de C) ni calculette

N'oubliez pas de commenter vos programmes.
Les types des paramètres et des fonctions sont parfois volontairement omis dans l'énoncé.
A vous de les définir.

Vous aurez besoin de 4 copies : une par exercice

1. Gestion d'une classe de CP (5 points) → copie n°1

Une maîtresse d'école souhaite informatiser la gestion des récompenses de ses élèves. Pour cela, elle a regroupé au sein d'une structure les informations concernant chacun de ses élèves, à savoir le nom, le prénom, le nombre de bons points et le nombre d'images. Pour 10 bons points, la maîtresse donne une image et pour cinq images, l'élève a droit à un cadeau (un livre).

1) Définir la structure *unEleve.*
2) Ecrire la fonction *initFichier()* qui initialisera en début d'année le fichier "cp.dat" avec le nom, prénom de chaque élève (rentrés par la maitresse) et mettra à 0 les autres champs.
3) Ecrire la fonction *ajouteUnBonPoint()* qui ajoute un bon point à un élève et met à jour les différents champs de la structure. Si l'élève a droit à un cadeau, la fonction renverra 1 sinon elle renverra 0. La fonction aura comme paramètre une variable de type *unEleve.*
4) Ecrire la fonction *MAJFichier()* qui demande un nom d'élève, recherche dans le fichier "cp.dat" l'élève et lui ajoute un bon point, la fonction affichera "cadeau" si l'élève mérite un cadeau. La fonction ne traitera pas la partie réécriture dans le fichier. On considéra que le nom rentré par la maitresse se trouve impérativement dans le fichier.

2. Fonctions et récursivité (4 points) → copie n°2

Composition de rotations à l'aide de quaternions

La composition de rotations dans l'espace en 3 dimensions est communément utilisée sous le formalisme des angles d'Euler (tangage, roulis, lacet), mais cette représentation possède un problème de singularité dans certains cas particuliers de compositions des rotations[1]. On peut alors utiliser le formalisme des quaternions qui ne possède pas ce problème. Un quaternion est un quadruplet de nombres réels, le premier élément étant un scalaire, et les trois éléments restants formant un vecteur. La composition des rotations dans ce formalisme consiste simplement à faire le produit des quaternions.

On va supposer qu'une représentation des 3 angles d'Euler est décrite dans une structure nommée *Euler* et qu'un quaternion est décrit dans une structure *Quaternion* composée de 4 éléments.

Soient les fonctions suivantes, supposées connues :

- *void EulerToQuaternion (Euler E, Quaternion *Q)* qui transforme les angles d'Euler (sous le type Euler) en quaternion
- *void QuaternionToEuler (Quaternion Q, Euler *E)* qui transforme un quaternion en angles d'Euler
- *void ProduitQuaternions (Quaternion Q1, Quaternion Q2, Quaternion *Q3) qui calcule le produit de deux quaternions*

1) Ecrire la fonction Compose (Euler E1, Euler E2, Euler *E3) qui calcule la composition de 2 rotations E1 et E2 en une rotation *E3 (*écrites toutes trois sous le formalisme d'Euler) en utilisant le produit de quaternions.

Fibonacci

Soit la suite de Fibonacci définie de la manière suivante :

$$u_0 = 1 \qquad u_1 = 1, \qquad u_n = u_{n-1}+u_{n-2} \quad n>1$$

2) Ecrire la fonction récursive *FiboRécursive(int n)* qui renvoie le n^{ième} terme de la suite de Fibonacci.

[1] Problème appelé « blocage de cardan »

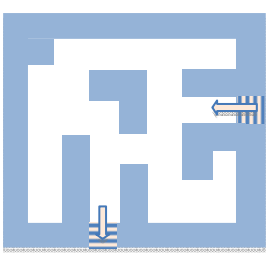
3. Fusion de tableaux (5 points) → copie n°3

1) Ecrire une fonction *SaisieTab()* permettant à l'utilisateur de saisir *N* réels dans un tableau qui peut contenir jusqu'à 100 éléments, $N \leq 100$. La fonction prendra en paramètre le tableau et retournera le nombre d'éléments (*N*) saisi par l'utilisateur;
2) Ecrire une fonction *FusionTab()* qui prend en entrée deux tableaux *A* et *B* (de dimensions respectives *N* et *M*), triés par ordre croissant, fusionne les éléments de *A* et *B* dans un troisième tableau *FUS* trié par ordre croissant. **On exploitera le fait que *A* et *B* sont déjà triés pour construire *FUS*.**
3) Ecrire une fonction *main()* qui fait appel aux deux fonctions précédentes et affiche le tableau *FUS* résultant. Le tableau *FUS* sera un tableau avec allocation dynamique.

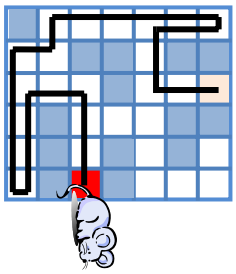
4. Souris dans un labyrinthe (6 points) → copie n°4

On veut simuler le parcours d'une souris dans un labyrinthe (fig 1.a). Le labyrinthe est représenté par une grille 2D de *nbL* lignes et *nbC* colonnes (fig 1.d). Chaque case de la grille est marquée comme libre (0) ou mur (1). Une case du bord est la porte d'entrée (2) et une autre la porte de sortie (3). Comme le montre la fig. 1.b, les murs extérieurs ne sont pas représentés explicitement dans la grille. Excepté pour l'entrée et la sortie du labyrinthe, chaque bord de la grille est considéré comme un mur.

On va placer une souris à l'entrée et lui faire parcourir la grille jusqu'à trouver la sortie. **La stratégie consiste à longer les murs par la droite**. Ainsi, à chaque étape elle va pivoter d'un quart de tour vers la droite, si c'est libre devant elle avance d'une case, sinon elle tourne d'un quart de tour vers la gauche et si c'est libre devant, elle avance sinon, elle refait un quart de tour vers la gauche et ainsi de suite. Dès qu'elle a pu avancer elle pivote vers la droite etc… Un exemple de parcours avec cette stratégie est donné dans la fig 1.c.



	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	0	0	1	1	0	0	0
2	0	0	0	1	0	0	2
3	0	1	0	0	0	1	1
4	0	1	0	1	0	1	0
5	0	1	3	1	0	0	0



```
typedef struct {
  int nbL ; /* nb de lignes */
  int nbC ; /* nb de colonnes */
  int lab[MAX][MAX]; /*grille 2D*/
 } unLabyrinthe ;
```

a.labyrinthe | b.représentation dans une grille | c.parcours dans le labyrinthe | d. type unLabyrinthe basé sur une structure

Figure 1. la souris dans le labyrinthe

On supposera connue la fonction *void initLabyrinthe(unLabyrinthe *L)* qui initialise un labyrinthe.
Soit une souris représentée par une position (ligne colonne) et une direction (S→sud, N→nord, O→ouest, E→ est). Par exemple la souris (1,4,'S') est en ligne d'indice 1, en colonne d'indice 4 et se dirige vers le sud.

1) Ecrire le type basé sur une structure *uneSouris*.
2) Ecrire la fonction qui initialise une souris à partir des données du labyrinthe. Avec l'exemple de la figure 1.b, la souris serait initialisée avec les valeurs de l'entrée soit (2,6,'O').
3) Ecrire la fonction *int avance(unLabyrinthe L, uneSouris *S)* qui fait avancer d'une case la souris dans sa direction, si la case qui se trouve devant la souris est libre. Dans ce cas la fonction retourne 1. Si la case qui se trouve devant la souris est un mur, la fonction retourne 0 et ne déplace pas la souris.
4) Supposons connues les fonctions *tourneADroite(uneSouris *S)* et *tourneAGauche(uneSouris *S)* qui font faire un quart de tour à droite ou à gauche à la souris. Ces 2 fonctions ne modifient pas la position de la souris mais uniquement sa direction.
   Ecrire le programme principal qui i) initialise le labyrinthe, ii) initialise la souris avec la case marquée comme « porte d'entrée » avec une direction allant vers l'intérieur du labyrinthe et iii) affiche le chemin parcouru par la souris sous forme de positions successives en appliquant la stratégie qui consiste à longer les murs par la droite.

Rappel : T[i][j] est la composante d'un tableau T à 2 dimensions en ligne i et colonne j.