Examen Programmation structurée Session 1

Année : 2018-2019

Tous documents interdits, calculatrice interdite

— Les questions sont indépendantes dans une large mesure. Vous pouvez utiliser les fonctions écrites dans les questions précédentes sans les réécrire.
— Si une question peut être interprétée de plusieurs manières, indiquez celle que vous avez choisie.
— Toutes les fonctions et programmes doivent être rédigés en langage C.
— Le barème est donné à titre indicatif.

1 Calcul de suite : racine carrée (2 pts)

La racine carrée du réel A peut être calculée comme la limite de la suite suivante :

$$\begin{cases} X_0 = & A \\ X_{i+1} = & \frac{(X_i + \frac{A}{X_i})}{2} \end{cases}$$

La limite de la suite est atteinte lorsque la différence relative entre 2 termes successifs de la suite X_i est inférieure à une valeur ϵ fixée par l'utilisateur. Le résultat est la dernière valeur calculée X_i.

1. Ecrire une fonction `double racine(double a, double eps)` qui calcule la limite de la suite ci dessus.
2. Ecrire un programme qui lit un nombre réel au clavier, calcule puis affiche sa racine carrée s'il est positif pour une valeur de `eps` égale à 10^{-7}

2 Paramètres de fonctions (4 pts)

3. Ecrire une fonction qui calcule la moyenne arithmétique de 2 réels **a** et **b**. Cette fonction prend 3 paramètres : les deux réels **a** et **b**, ainsi qu'une variable pour stocker la moyenne. **La fonction ne retourne aucune valeur**. Cette fonction ne fait aucun affichage à l'écran.
4. Ecrire une fonction qui calcule et retourne le produit de 2 réels **a** et **b**. Les deux réels **a** et **b** sont passés en paramètres. La fonction **retourne** le produit et ne fait aucun affichage à l'écran.
5. Ecrire une fonction qui calcule la moyenne arithmétique et le produit de 2 réels **a** et **b**. Cette fonction prend 4 paramètres : les deux réels **a** et **b**, ainsi que deux variables pour stocker les deux résultats (moyenne et produit). Cette fonction ne fait aucun affichage à l'écran. **Vous devez utiliser les deux fonctions précédentes** pour faire cette fonction.
6. Ecrire un programme qui lit 2 réels au clavier, calcule la moyenne et le produit de ces 2 réels en utilisant la fonction précédente, puis affiche ces valeurs.

3 Allocation dynamique et Image (9 pts)

La structure `IMAGEUCHAR` permettant de manipuler des images est définie de la manière suivante :

```
// Definition du type PIXEL
typedef unsigned char  PIXEL;

// Definition du type IMAGEUCHAR pour des images 8 bits d'octets
typedef struct {
  int nl,nc;
  PIXEL** val;
} IMAGEUCHAR;
```

- Les champs `nl` et `nc` désignent respectivement les nombres de lignes et de colonnes de l'image.
- Le champ `val` est un tableau 2D (une matrice) de valeurs sur 8 bits (les pixels) qui doit être alloué dynamiquement en respectant une allocation contigüe. Après cette allocation, il est donc possible pour une variable `im` de type `IMAGEUCHAR` d'utiliser la notation `im.val[i][j]` pour accéder au pixel de coordonnées `i,j`. L'allocation contigüe permet de voir aussi l'image comme un tableau à 1 dimension, dont l'adresse de départ est `im.val[0]` ou `*(im.val)`.

7. Ecrire la fonction qui réalise l'allocation dynamique de l'espace mémoire pour le tableau 2D de pixels de `nbLigne` lignes et `nbColonne` colonnes. Elle retourne le tableau alloué ou `NULL` si l'allocation est impossible. Son prototype est :
   `PIXEL** alloueMemoirePixel(int nbLigne, int nbColonne);`
8. Ecrire la fonction qui initialise les 3 champs `nl,nc,val` et retourne une structure `IMAGEUCHAR` pour une image de `nbLigne` lignes et `nbColonne` colonnes. Elle initialise le champ `val` avec une allocation dynamique contigüe en utilisant la fonction précédente. Si l'allocation est impossible, les valeurs des 2 champs `nl` et `nc` seront mises à `ZERO`, celle du champ `val` sera mise à `NULL`. Son prototype est :
   `IMAGEUCHAR creationImageUChar(int nbLigne, int nbColonne) ;`
9. Une image n'est pas vide si le nombre de lignes et le nombre de colonnes sont strictement positifs. Ecrire la fonction qui teste si l'image `im` est vide. Elle retourne `0` si l'image n'est pas vide, `1` sinon. Son prototype est :
   `int estVideImageUChar(IMAGEUCHAR im) ;`
10. Ecrire la fonction qui met à la valeur `valeur` toutes les valeurs des pixels de l'image `im` si elle existe. Son prototype est :
    `void setImageUChar(IMAGEUCHAR im, PIXEL valeur) ;`
11. Ecrire la fonction qui ré-initalise l'image pointée par le paramètre `pim`. Elle met à `ZERO` les 2 champs `nl,nc`. Elle libère l'espace mémoire alloué pour le tableau de pixels si nécessaire puis met le champ `val` à `NULL`. Son prototype est :
    `void libereImageUChar(IMAGEUCHAR* pim);`
12. Ecrire la fonction qui teste si le contenu de 2 tableaux de `PIXEL t1` et `t2` à 1 dimension de même taille `nbOctet` sont identiques. Pour cela, il faut parcourir les 2 tableaux simultanément et quitter la fonction avec la valeur `0` si les 2 valeurs examinées ne sont pas égales. Lorsque la fin du tableau est atteinte, les contenus sont identiques et la fonction retourne `1`. Cette fonction sera écrite en utilisant **des indices pointeurs uniquement** et une notation du style `*p1` ou `*p1++`. Il faut donc définir 2 pointeurs pour parcourir les 2 tableaux. Il est **interdit d'utiliser un indice entier** `i` et une notation du style `t[i]`. Son prototype est :
    `int egalTableauUChar(PIXEL* t1, PIXEL* t2, int nbOctet);`
13. Ecrire la fonction qui teste si 2 images sont identiques. Deux images sont identiques si elles ont le même nombre de lignes et de colonnes et si les tableaux de pixels comportent les mêmes valeurs. Elle retourne `1` si les images sont identiques, `0` sinon. Utilisez **obligatoirement** la fonction précédente. Son prototype est :
    `int egaleImageUChar(IMAGEUCHAR im1, IMAGEUCHAR im2);`

4 Image et Template Matching (3 pts)

Le Template matching consiste à calculer la ressemblance, en chaque pixel, entre une image `I` et un modèle `T`. On calcule une distance entre un modèle `T` de dimension `nlm,ncm` et la portion de l'image `I` centrée en `i,j`, comprise dans le rectangle de coordonnées `(i-nlm/2,j-ncm/2)` et `(i+nlm/2,j+ncm/2)`. La distance choisie entre le modèle `T` et la partie de l'image `I` centrée en `i,j` est simplement **l'écart absolu moyen** :

$$d_{T,I}(i,j) = \frac{1}{nlm.ncm}. \sum_{di=0}^{nlm-1} \sum_{dj=0}^{ncm-1} |I[i-nlm/2+di][j-ncm/2+dj] - T[di][dj]| \quad (1)$$

Cette distance est comprise entre `ZERO` (l'image `I` en `i,j` correspond exactement au modèle `T`) et `255`.

14. Ecrire la fonction qui calcule la distance `d` entre le modèle `modele` et l'image `im` au pixel de coordonnées `is, js`. Dans cette fonction, les valeurs des paramètres `is` et `js` sont telles que le calcul de la distance (eq 1) ne fait pas intervenir des pixels en dehors de l'image (c.f. ci dessous). Son prototype est :
    `unsigned char distance( IMAGEUCHAR im, IMAGEUCHAR modele, int is, int js);`
15. Ecrire la fonction qui calcule la distance `d` entre le modèle `modele` et l'image `im` pour **tous** les pixels de l'image. Lorsque le calcul de distance (eq 1) fait intervenir des pixels en dehors de l'image, leur valeur de distance sera égale à `255`. On pourra initialiser l'image des distances avec la valeur 255 et ne calculer la distance que pour les pixels pour lesquels ce calcul est possible. Cette fonction retourne une image dont les valeurs sont les distances entre l'image `im` et le modèle `modele` en chaque point. Son prototype est :
    `IMAGEUCHAR templateMatching(IMAGEUCHAR im, IMAGEUCHAR modele) ;`
16. Quel est le nombre d'opérations arithmétiques (multiplication, addition et soustraction) effectuées par la fonction `templateMatching` en fonction des tailles de l'image et du modèle ?

5 Image et lecture dans un fichier (3 pts)

Le format de fichier image `PGM` texte est un format pour stocker des images en niveaux de gris dans un fichier. Ce format de fichier simplifié pour cet examen comporte deux parties distinctes :

- un entête contenant des informations en mode texte, que l'on peut lire facilement avec les fonctions `fscanf`, `fgets`, `fgetc`. Il n'y aura aucune autre information dans cet entête que celles décrites ci dessous, contrairement à la norme officielle.
  - Une ligne contenant le `magic number` c'est à dire la chaîne de caractères "P2".
  - Une ligne contenant le nombre de colonnes puis le nombre de lignes
  - Une ligne contenant la plus grande valeur possible d'intensité, inférieure ou égale à 255
- les données images en mode texte
  Les valeurs de tous les pixels en ASCII sont rangées dans l'ordre d'apparition sur l'image par un balayage vidéo de gauche à droite, de haut en bas. Elles doivent donc être lues avec la fonction `fscanf`. Elles ont été écrites avec la fonction `fprintf`.

17. Ecrire la fonction de lecture d'une image pointée par le paramètre `pim` dans un fichier de nom `fic`. Son prototype est :
    `int lectureImagePgmTexte(char* fic,IMAGEUCHAR * im) ;`
    Cette fonction réalise les actions suivantes
    - Ouvre le fichier
    - Lit le Magic Number. Retourne `1` si ce n'est pas la bonne chaîne de caractères.
    - Lit le nombre de colonnes et de lignes.
    - Crée une nouvelle image à l'aide des fonctions précédentes. Quitte la fonction si l'allocation n'est pas possible avec la valeur `2`.
    - Lit la valeur maximale des intensités. Si la valeur est supérieure à `255`, libère l'espace alloué et quitte la fonction avec la valeur `3`.
    - Lit les valeurs de pixels et les stocke dans l'image pointée par le paramètre `pim`. Quitte la fonction avec la valeur 4 si la valeur d'un pixel ne peut être lue.
    - Quitte la fonction avec la valeur `0`.

6 Programme (2 pts)

18. Ecrire un programme qui réalise les actions suivantes :
    - Définit les variables utiles
    - Lit une image à partir du fichier "image1.pgm" au format `PGM`.
    - Lit un modèle à partir du fichier "modele1.pgm" au format `PGM`.
    - Calcule l'image des distances entre l'image et le modèle
    - Libère les images utilisées

7 Résumé de la syntaxe du langage C

Structure d'un programme

```
int main() {
  declaration des variables ;
  instructions ;
}
```

Déclaration des variables type identifiant ;

```
 int i ; double x ; char c ; char s1[256] ; int* p1 ; double* p2 ;
```

Entrées fonction scanf

```
scanf("%d␣",&i) ; scanf("%lf", &x) ; scanf("%c", &c) ; scanf("%s", s1) ;
```

Sorties fonction printf

```
printf("%d␣",i) ; printf("%lf", x) ; printf("%c", c) ; printf("%s", s1) ;
```

Boucle POUR

```
for (<initialisation>; <test de continuation>; <incrementation>) {
    <instructions a repeter> ;
}
for (i=0 ; i< 100 ; i++) { t1[i] = 2*i ; }
```

Boucle TANTQUE

```
  <initialisation>
  while (<test de continuation>) {
    <instructions a repeter> ;
    <incrementation> ;
  }
while ( i< 100 ) { t1[i] = 2*i ;  i= i+1 ;}
```

Boucle FAIRE..TANT_QUE

```
  <initialisation>
  do {
    <instructions a repeter> ;
    <incrementation> ;
  } while (<test de continuation>) ;
do { t1[i] = 2*i ;  i= i+1 ;} while ( i< 100 ) ;
```

Instruction condionnelle instruction if

```
if (test) { instructions si test  est vrai ; }
else { instructions si test est faux; }
if (a !=0) { c = b/a ; }
else { printf("La␣division␣est␣interdite\n") ; }
```

Pointeurs et adresse : l'opérateur & permet d'obtenir l'adresse d'une variable, l'opérateur * permet d'obtenir la valeur située à une adresse ;

```
int i ;
int* p =NULL; /* p est une variable pointeur contenant une adresse*/
p=&i ;        /* p contient maintenant l'adresse de i*/
*p = 9 ;      /* On met 9 dans la variable situee a l'adresse donnee par p,
                 ie on met 9 dans i*/
```

Fonctions : déclaration

```
type  Identifiant_fonction(declaration des parametres de la fonction) {
  Declaration des variables ;
  Instructions de la fonction ;
}
Exemple :
double max( double a, double b) { double c ;
```

```
  if (a>b) c =a ;
  else c=b ;
return c ;
}
```

Tableau 1D : type identifiant_tableau[Nombre d'element] ; L'identifiant ou nom du tableau est son adresse.
L'accès à un élément de tableau s'écrit t[i] ou *(t+i).
L'adresse d'un élément du tableau s'écrit &t[i] ou t+i.

```
int t1[100] ; double x1[256] ;
```

Tableau 2D : type identifiant_[Nombre d'element][Nombre d'element] ;

```
Exemple : int t2d[100] [20]; double x2d[256] [3];
```

Structure Déclaration d'un type structure : struct identifiant_structure { déclarations des champs ; } ; Déclaration d'une variable de type structuré : struct identifiant_structure variable ;

```
struct complex { double reel ; double imaginaire ; } ; /* Un type complexe*/
struct complex x ; /* x est une variable de type complexe */
```

Déclaration d'un nouveau type : typedef ancien_type nouveau_type ;

```
typedef struct complex MesComplexes ;
```

Fichiers — Declaration d'une variable : FILE* identifiant_variable ;

— Ouverture :
`FILE* fopen(char* nom_du_fichier_a_ouvrir, char* mode_d'ouverture) ;`

— Lecture fichier texte :
`int fscanf(FILE* variableFichier, char* format_des_données_a_lire, adresses_des_données);`

— Ecriture fichier texte :
`int fprintf(FILE* variableFichier, char* formatVariables_a_ecrire , variables_a_ecrire);`

— Lecture fichier binaire :
`int fread(void* adresse_donnees, int taille_d'un_element, int nombre_d'element, FILE* variable_fi`
`dans lequel on lit) ;`

— Ecriture fichier binaire :
`int fwrite(void* adresse_donnees, int taille_d'un_element, int nombre_d'element, FILE* variable_f`
`dans lequel on ecrit) ;`

Gestion mémoire Allocation dynamique

— Allocation d'un espace :
`void* calloc(int nombre d'element a allouer, int taille d'un element) ;`

— Libération mémoire :
`void free(void* adresse de la zone a liberer) ;`

Quelques fonctions sur les chaînes de caractères et les copies mémoires

— Lecture d'une chaîne :
`scanf("%s",mot)` : lecture d'une chaîne au clavier. Remplace le caractère '\n' (« retour chariot ») par le caractère '\0' (marque de fin de chaîne).

— Longueur d'une chaîne
`int strlen(char* mot)` : retourne le nombre de caractères de la chaîne

— Comparaison de 2 chaînes :
`int strcmp(char* s1, char* s2)` : retourne une valeur négative si s1<s2, nulle si s1 et s2 sont identiques, et positive si s1>s2 : c'est l'ordre lexicographique qui est utilisé.

— Copie d'une chaîne :
`char* strcpy(char* s1, char* s2)` : copie de la chaîne s2 dans la chaîne s1. La chaîne s1 doit comporter suffisamment de place pour contenir la chaîne s2.

— Copie d'un tableau :
`void* memcpy(void* dst, void* src, int n)` : copie n octets du tableau src dans le tableau dst.